М. С. Таірова, З. Ю. Журавльова

МОВА ПРОГРАМУВАННЯ PYTHON ДЛЯ НАУКОВИХ ОБЧИСЛЕНЬ

Частина 1

НАВЧАЛЬНИЙ ПОСІБНИК

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І. І. МЕЧНИКОВА
ФАКУЛЬТЕТ МАТЕМАТИКИ, ФІЗИКИ і ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

М. С. Таірова, З. Ю. Журавльова

МОВА ПРОГРАМУВАННЯ PYTHON ДЛЯ НАУКОВИХ ОБЧИСЛЕНЬ

Частина 1

НАВЧАЛЬНИЙ ПОСІБНИК

з дисципліни «Програмні засоби наукових обчислень»
для студентів спеціальності 113 «прикладна математика»

ОДЕСА
ОНУ
2022

УДК 519.85:004.43
Т145

Автори:

М. С. Таірова, к.ф.-м.н., доц. кафедри математичного та комп'ютерного моделювання (на 2022 год) (с 01.09.2022 – кафедра оптимального керування і економічної кібернетики) Одеського національного університету імені І. І. Мечникова;

З. Ю. Журавльова, к.ф.-м.н., доц. кафедри методів математичної фізики Одеського національного університету імені І. І. Мечникова.

Рецензенти:

А. А. Кобозєва, д. т. н., проф., зав. кафедри кібербезпеки та програмного забезпечення Державного університету «Одеська політехніка»;

В. В. Пічкур, д. ф.-м. н., проф. кафедри моделювання складних систем факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка.

Рекомендовано до друку вченою радою
ОНУ імені І. І. Мечникова.
Протокол № 3 від 26.10.2021 р.

Таірова М. С.

Т145 Мова програмування Python для наукових обчислень. Частина 1 : навч. посіб. з дисципліни «Програмні засоби наукових обчислень» / М. С. Таірова, З. Ю. Журавльова. – Одеса : Одес. нац. ун-т ім. І. І. Мечникова, 2022. − 262 с.

ISBN 978-617-689-533-6

Посібник містить основні теоретичні відомості до тем, які супроводжуються контрольним завданням, котре передбачає самостійну роботу студентів при вивченні даного курсу.

Для студентів напряму 113 «Прикладна математика».

УДК 519.85:004.43

ISBN 978-617-689-533-6

Зміст

Вступ

Python (часто вимовлятися як пайтон або пітон) – високорівнева мова програмування загального призначення, орієнтована на підвищення продуктивності розробника і читання коду. Синтаксис ядра Python мінімалістичний. У той же час стандартна бібліотека включає великий обсяг корисних функцій.

Історія мови Python починається в 1980-х роках, коли Гвідо ван Россум, співробітник центру математики та інформатики в Нідерландах, приступив до створення його першої версії. Гвідо, творець мови Python, назвав його так на честь телешоу на BBC під назвою «Monty Python's Flying Circus» («Літаючий цирк Монті Пайтона»). І до сих пір «Гвідо ван Россум» залишається «великодушним довічним диктатором». Слід зазначити, що Python – мова програмування, що активно розвивається, нові версії (з додаванням / зміною мовних властивостей) виходять приблизно раз в два з половиною роки.

До даного моменту були випущені вже три версії мови :

- Python 1 (1994 рік); - Python 2 (2000 рік); - Python 3 (2008 рік);

причому перші дві з них назад сумісні, а Python 3 і Python 2 – вже ні. Це пов'язано з тим, що ті поліпшення, які були зроблені в Python 3, неможливо внести без порушення сумісності. Детальніше про ключові відмінності Python 3 і Python 2 можна прочитати на сайті:

http://python.cx/blog/article/key_differences_python2.7.x_python3.x

Особливості мови та основні переваги мови Python:

1. Вільна ліцензія. Python випускається під вільною ліцензією «Python Software Foundation License», що дозволяє використовувати вихідний код проекту не тільки у відкритому, але і в комерційному програмному забезпеченні.

2. Бібліотека. Python має багату стандартну бібліотеку, і не тільки, крім неї існує велика колекція додаткових наукових бібліотек і середовищ. Виділимо наступні:

• **numpy** – пакет для матричних обчислень;
• **matplotlib** – бібліотека для побудови 2D і 3D графіків функцій;
• **pandas** – пакет для статистичної обробки даних;
• **sympy** – бібліотека символьних обчислень;
• **scipy** – пакет для наукових і інженерних розрахунків.

3. Широка область застосування:

• Системне програмування;

• Розробка програм з графічним інтерфейсом;
• Розробка динамічних веб-сайтів;
• Інтеграція компонентів;
• Розробка програм для роботи з базами даних;
• Швидке створення прототипів;
• Розробка програм для наукових обчислень;
• Розробка ігор;

4. Зрозуміла і простий мова для прочитання коду і навчання. Основною перевагою є простота програмування, зводячи до мінімуму час, необхідний для розробки, налагодження і підтримки коду. Ключова ідея Гуїдо така: код читається набагато більше разів, ніж пишеться. Власне, рекомендації про стиль написання коду спрямовані на те, щоб поліпшити читабельність коду і зробити його узгодженим між великим числом проектів. В ідеалі, весь код буде написаний в єдиному стилі, і будь-хто зможе легко його прочитати.

5. Легкий для навчання: У Python'а мало ключових слів, проста структура і чітко визначений синтаксис. Завдяки цьому навчитися основам мови можна за досить короткий час.

• Легко читається: Блоки коду в Python виділяються за допомогою відступів, що спільно з ключовими словами, взятими з англійської мови значно полегшує читання коду.

• Легкий в обслуговуванні: Однією з причин широкої популярності Python є простота обслуговування коду, написаного на цій мові.

• Наявність інтерактивного режиму: дозволяє "на льоту" тестувати потрібні ділянки коду.

• Портативність: Python без проблем запускається на різних платформах, при цьому зберігає однаковий інтерфейс, незалежно від того, на якому комп'ютері ви працюєте.

• Можливість розширення: при необхідності в Python можна впроваджувати низькорівневі модулі, написані іншими мовами програмування для найбільш гнучкого вирішення поставлених завдань.

• Робота з базами даних: в стандартній бібліотеці Python можна знайти модулі для роботи з більшістю комерційних баз даних.

• Створення GUI (Графічного інтерфейсу користувача): на Python можливе створення GUI додатків, які будуть працювати незалежно від типу вашої операційної системи.

Слід також виділити **недолік мови Python –** швидкість. Оскільки Python є інтерпретована і динамічно універсальна мова програмування, виконання коду на Python може бути повільним у порівнянні зі статично типізованими мовами програмування, такими як C і Fortran. Але слід зазначити, що стосується динамічних «побратимів» (PHP, Ruby, JavaScript), то Python у більшості випадків виконує код швидше за рахунок попередньої компіляції в байт-код і значної частини стандартної бібліотеки, написаної на C. Зазначимо, що цей недолік нівелює себе у випадках, коли перевага в швидкості розробки важливіше втрати швидкості виконання програми, особливо, якщо врахувати швидкодію сучасних комп'ютерів (ціна роботи людини і комп'ютера). Проте, навіть при високій швидкодії сучасних процесорів залишаються такі галузі, де потрібна максимальна швидкість виконання.

Phyton: сучасні додатки

• Компанія Google широко використовує Python у своїй пошуковій системі і оплачує працю творця.

• Служба колективного використання відеоматеріалів YouTube в значній мірі реалізована на мові Python.

• Популярна програма itTorrent для обміну файлами в пірінгових мережах (peer-to-peer) написана мовою Python.

• Популярний веб-фреймворк App Engine від компанії Google використовує Python в якості прикладної мови програмування.

• Такі компанії як EVE Online і Massively Multiplayer Online Game (MMOG) широко використовують Python в своїх розробках.

• Потужна система тривимірного моделювання і створення мультиплікації Maya підтримує інтерфейс для управління сценаріїв на мові Python.

• Такі компанії як Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm і IBM використовують Python для тестування апаратного забезпечення.

• Такі компанії як Industrial Light & Magic, Pixar та інші використовують Python у виробництві анімаційних фільмів.

• Компанії JPMorgan Chase, UBS, Getco і Citadel застосовують Python для прогнозування фінансового ринку.

• NASA, Los Alamos, Fermilab, JPL і інші використовують Python для наукових обчислень.

• iRobot використовує Python в розробці комерційних роботизованих пристроїв.

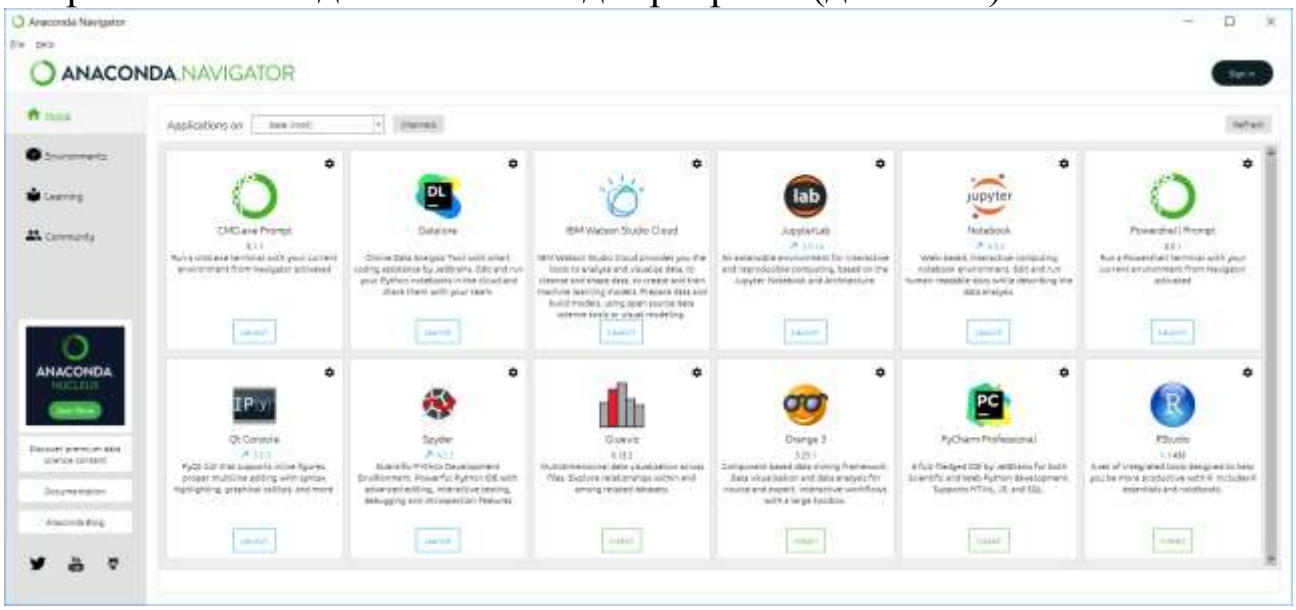
• ESRI використовує Python в якості інструменту налаштування своїх популярних геоінформаційних програмних продуктів під потреби кінцевого користувача.
• NSA використовує Python для шифрування і аналізу розвідданих.
• У реалізації поштового сервера IronProt використовується понад 1 мільйон рядків програмного коду на мові Python.
• Проект «ноутбук кожній дитині» (One Laptop Per Child, OLPC) будує свій користувальницький інтерфейс і модель функціонування на мові Python.

Основи Python
І Попередні відомості
1. Anaconda

Завантажується з сайту (версія Python 3):

https://www.anaconda.com/download/

До її складу входять, серед інших, Jupyter Notebook, Spyder. Основні бібліотеки вже завантажені, для їхнього використання в програмі їх потрібно лише підключити в коді програми (див. XI.1).

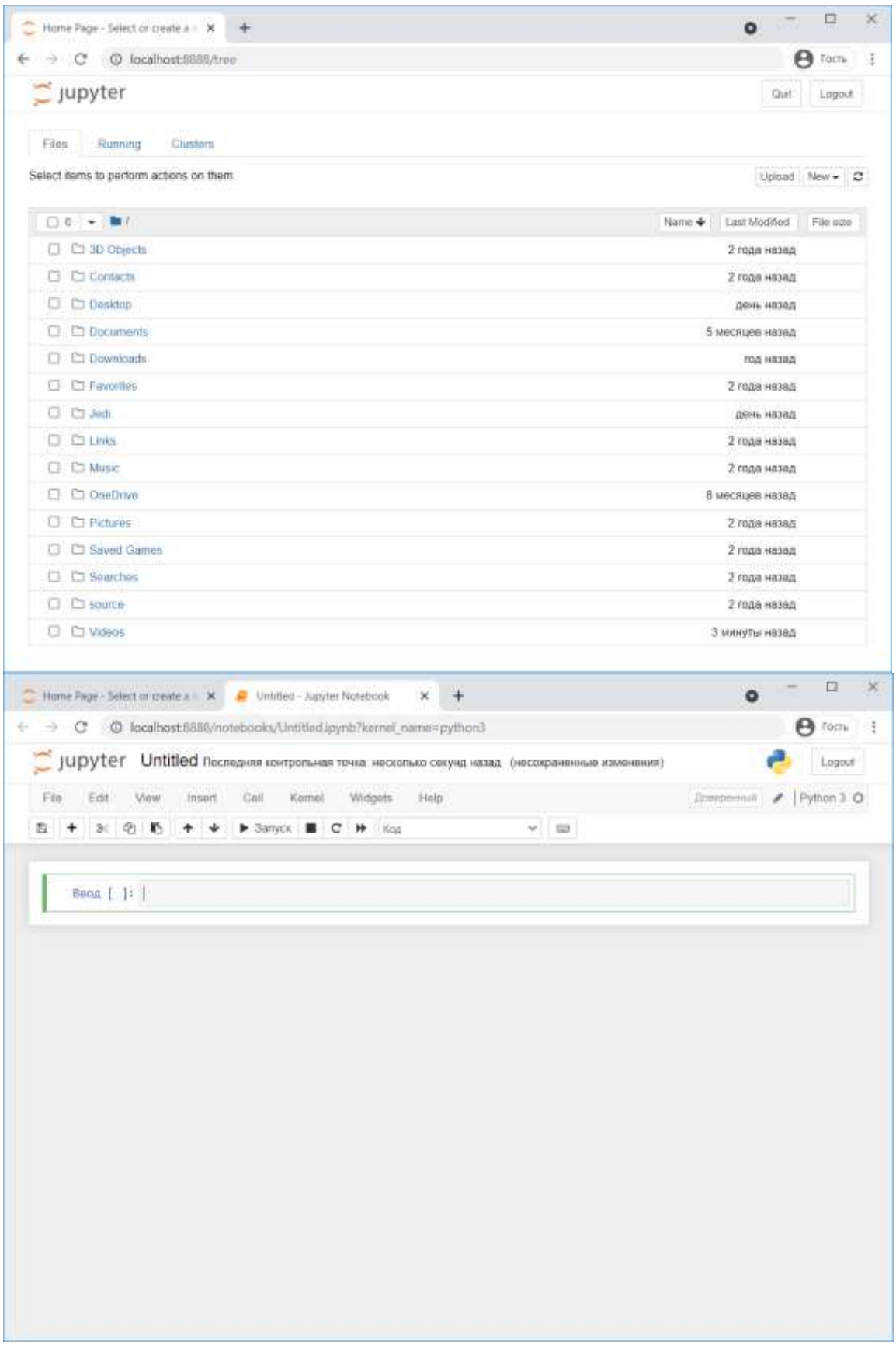


a. Jupyter Notebook

Код пишеться у клітинках, потім запускається і продовження пишеться в наступних клітинках.

Підтримується html-код.

Можна використовувати для створення презентацій.

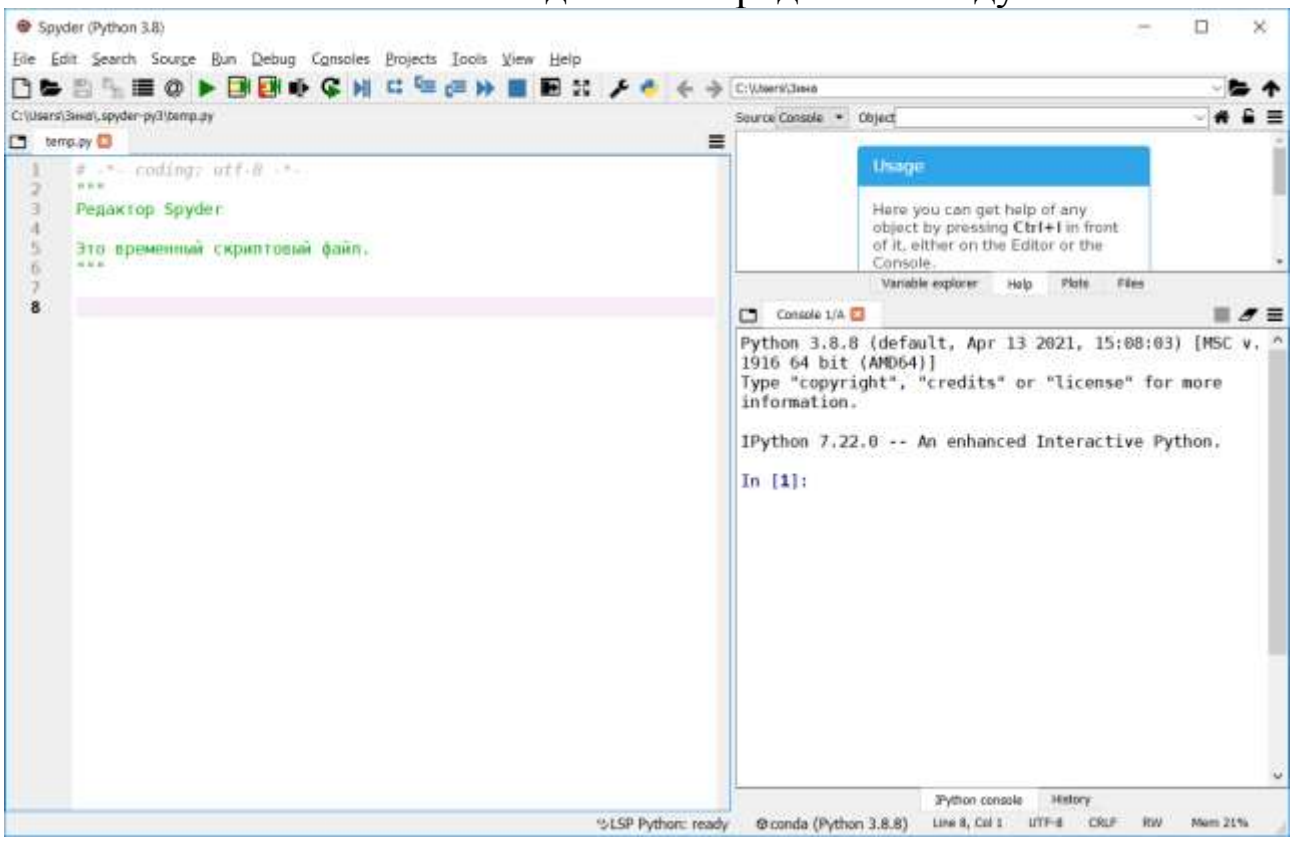

Home Page - Select or create a
localhost:8888/tree
Гость
jupyter
Quit
Logout
Files
Running
Clusters
Select items to perform actions on them.
Upload
New
Name
Last Modified
File size
3D Objects 2 года назад
Contacts 2 года назад
Desktop день назад
Documents 5 месяцев назад
Downloads год назад
Favorites 2 года назад
Jedi день назад
Links 2 года назад
Music 2 года назад
OneDrive 8 месяцев назад
Pictures 2 года назад
Saved Games 2 года назад
Searches 2 года назад
source 2 года назад
Videos 3 минуты назад
Untitled - Jupyter Notebook
localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3
jupyter Untitled Последняя контрольная точка: несколько секунд назад (несохраненные изменения)
Logout
File Edit View Insert Cell Kernel Widgets Help
Python 3
Запуск
Код
Ввод [ ]:

b. Spyder

Інтегроване середовище розробки для наукових обчислень з використанням Python. Найбільш зручна програма, що складається з редактора та консолі.

У **редакторі** можна писати і редагувати, запускати та зберігати у файлах *.py. Запуск – клавіша F5 або відповідна кнопка на панелі задач.

У **консолі** можна писати код і запускати, але редагувати лише поточну команду. Для запуска коду в консолі використовується клавіша Enter або Shift+Enter для багаторядкового коду.



1. Коментарі

У Python є 2 типи коментарів:

1) однорядковий;
2) багаторядковий.

Для багаторядкових коментарів використовуються одинарні або подвійні лапки.

Приклад

```
#  це однорядковий коментар

'''це багаторядковий коментар
'''
```

"""
це теж багаторядковий коментар
"""

2. Виведення даних – кратко

Для виведення даних використовується функція print. За її допомогою можна виводити як строкові дані, так і дані інших типів. Якщо потрібно вивести декілька різних величин, які, наприклад, збережено у різних змінних, то їх можна розділити комою у функції print. Більш детально ця функція розглянута у V.2.

Приклад

print('Text')
```
Text
```

print('Text1', 'text2')
```
Text1 text2
```

3. Змінні

Python є строго об'єктно-орієнтованою мовою програмування, тому все у Python є ***об'єктом***, включаючи числа, строки, функції.

Об'являти змінні у Python не потрібно, вони використовуються простим присвоюванням їм значень. По присвоєному значенню автоматично визначається тип змінної. У процесі роботи змінній можуть бути присвоєні значення різних типів, відповідно, автоматично буде змінюватися тип змінної.

4. Імена змінних

Імена змінних можуть починатися з букви одного з алфавітів Unicode або символу нижнього підкреслювання. Інша частина імені може містити також цифри. Імена змінних є чутливими до регістру.

Приклад

variable=5
print(variable)
```
5
```

змінна=5
print(змінна)
```
5
```

```
_var=6
__var=7

var1=5
перем1=5

var1=5
Var1=10
print(var1,Var1)
5 10
```

5. Запис інструкцій

У Python кожна інструкція пишеться з нового рядка, крапка з комою в кінці не є обов'язковою. Однак, якщо потрібно записати декілька інструкцій в одному рядку, їх можна розділити крапкою з комою.

Приклад

```
a=1; b=5; c=-3
```

Можна записати одну інструкцію у декількох строках. Для цього достатньо заключити її у пару круглих, квадратних або фігурних дужок відповідно до типу даних. Або можна поставити знак \ наприкінці перших строк.

Приклад

```
d=((a+b*c)/
 (a**2-5*b+c/a))

d=(a+b*c)/ \
 (a**2-5*b+ \
 c/a)
```

6. Зауваження до прикладів

Приклади виконано в консолі IPython додатку Spyder, тому для виведення результату не завжди використовується функція print.

II Базові типи даних

Python є **динамічно типізованою** мовою, тому не має необхідності визначати тип змінних, аргументів функцій та типів повернутих даних.

Python підтримує **автоматичне управління пам'яттю**: не має необхідності явно виділяти та звільняти пам'ять для змінних та масивів даних. Не має помилок витоку пам'яті.

Базові типи даних можна поділити на:

1) Числові
    a. цілі int;
    b. дійсні float;
    c. комплексні complex;
2) Логічні bool.

1. Числові дані

a. Цілі числа

Функція type повертає тип змінної.
Python підтримує довгі цілі числа.

Приклад

```
x=1
type(x)
Out[1]: int

x=12345678901234567890123456789O

x*x
Out[21]:
1524157875323883675049535156253619878750190519987501905210O
```

b. Дійсні числа

Роздільником між цілою та дробовою частиною є крапка. Довгі дійсні числа не підтримуються, вони приводяться до стандартного уявлення дійсних чисел у вигляді мантиси та показника.

Приклад

```
x=1.0
type(x)
Out[1]: float

x=2.5E-5
x
Out[2]: 2.5e-05
```

x=5.75**E**+3 #$5.75 \cdot 10^3$
x
Out[3]: 5750.0

x=12345678901234567890123456789**0.0**
x
Out[4]: 1.2345678901234568e+29

x*x
Out[5]: 1.5241578753238835e+58

c. Комплексні числа

Python підтримує комплексні числа, уявна частина позначається символом j.

Для виділення дійсної та уявної частин числа використовуються параметри real та imag.

Для отримання комплексно-спряженого числа використовується функція conjugate.

Приклад

x=1.0-1.0**j**
x
Out[1]: (1-1j)

type(x)
Out[2]: complex

x=1.0-1.0j
print(x.**real**, x.**imag**)
1.0 -1.0

x.**conjugate**()
Out[3]: (1+1j)

d. Перевірка типів

Для перевірки того, чи є тип змінної x дійсним, можна скористатися одним з 2 способів:

1) type(x) is float
2) isinstance(x, float)

Аналогічно можна здійснити перевірку для будь-якого іншого типу даних.

Приклад

```
x=1.0
type(x) is float
Out[1]: True

type(x) is int
Out[2]: False

isinstance(x, float)
Out[3]: True

print(x, type(x))
1.0 <class 'float'>
```

Для приведення до типу float використовується однойменна функція. Аналогічні функції є для усіх базових типів даних.

```
x=1
y=float(x)

y
Out[1]: 1.0

z=int(y)
z
Out[2]: 1
```

е. Системи числення

У Python є декілька функцій для перетворення цілого числа в інші системи числення, а саме:

- bin(x) – для перетворення десяткового числа x у 2-ічну строку;
- hex(x) – для перетворення десяткового числа x у 16-ічну строку;
- oct(x) – для перетворення десяткового числа x у 8-ічну строку.

Приклад
x=23

bin(x)
Out[1]: '0b10111'

hex(x)
Out[2]: '0x17'

oct(x)
Out[3]: '0o27'

Також за допомогою функції int(x, y) можна перевести число x, задане строкою, у десяткову систему числення з y-ічної.

Приклад
int('10111', 2)
Out[1]: 23

int('17', 16)
Out[2]: 23

int('27', 8)
Out[3]: 23

2. Логічні дані

У Python є дві логічні сталі True та False, що відповідають за істину та хибність відповідно.

Приклад
b1=True
b2=False

type(b1)
Out[1]: bool

III Оператори

Оператори у Python можна поділити на:

- арифметичні;
- логічні;

- оператори порівняння;
- присвоювання.

1. Арифметичні оператори

У Python є наступні алгебраїчні оператори:

- + складення;
- - віднімання;
- * множення;
- / ділення;
- // ділення націло;
- % залишок від ділення націло;
- ** зведення у ступінь.

Приклад

```
1+2, 1-2, 1*2, 1/2
Out[1]: (3, -1, 2, 0.5)

3.0//2.0
Out[2]: 1.0

3%2
Out[3]: 1

2**2
Out[4]: 4
```

Результат операції ділення / для цілих чисел залежить від версії Python. Так, у версіях Python 3.x результат буде дійсним числом, а у версіях Python 2.x – цілим.

Приклад

```
1/2
Out[1]: 0.5 #(float) в версиях Python 3.x

1/2
Out[2]: 0 #(int) в версиях Python 2.x
```

2. Логічні оператори

У Python є наступні логічні оператори:

- and – логічне І – кон’юнкція;

- or – логічне АБО – диз'юнкція;
- not – логічне заперечення.

Приклад

True **and** False
Out[1]: False

not False
Out[2]: True

True **or** False
Out[3]: True

Пріоритети логічних операторів у порядку спадання:

- заперечення;
- кон'юнкція;
- диз'юнкція.

Приклад

x=True; y=False; z=True

not x and y or z
Out[1]: True

((not x) and y) or z
Out[2]: True

3. Оператори порівняння

У Python є наступні оператори порівняння:

- <, >
- <=, >=
- ==, !=

Приклад

2>1, 2<1
Out[1]: (True, False)

2>2, 2<2, 2>=2, 2<=2
Out[2]: (False, False, True, True)

2==2, 2!=2

Out[3]: (True, False)

4. Присвоювання

У Python відсутній оператор ++ та подібні, проте, є оператори типу +=.

Приклад

```
x=2

x+=1
x
Out[1]: 3

x-=2
x
Out[2]: 1

x*=5
x
Out[3]: 5

x/=2
x
Out[4]: 2.5
```

У Python можна присвоювати одне значення декільком змінним відразу.

Приклад

```
x=y=z=5
```

5. Пріоритети операторів

Пріоритети операторів від самого низького до самого високого подано у наступній таблиці.

Таблиця 1

Оператор	Опис
lambda	лямбда-вирах
or	логічне «АБО»
and	логічне «І»

not x	логічне «НІ»
in, not in	перевірка приналежності
is, is not	перевірка тотожності
<, <=, >, >=, !=, ==	порівняння
|	побітове «АБО»
^	побітове виключне «АБО»
&	побітове «І»
<<, >>	зсуви
+, -	складання та віднімання
*, /, //, %	множення, ділення, ділення націло, залишок від ділення націло
+x, -x	позитивне, від'ємне
~x	побітове «НІ»
**	зведення у ступінь
x.attribute	посилання на атрибут
x[index]	звернення по індексу
x[index1:index2]	зріз
f(arguments…)	виклик функції
(expressions, …)	зв'язка або кортеж
[expressions, …]	список
{key: data, …}	словник

6. Завдання та вправи

1. Розставити дужки у відповідності до пріоритетів операторів:
   - 2**x+5<3*x-8//5
   - x and y or not x
   - x**2*x-8>=x%3**2

IV Складені типи даних

1. Строки

Строка – це послідовність символів Unicode. Типу char у Python не має.

Для завдання строк можна використовувати як одинарні, так і подвійні лапки, але обов'язково однакові. Для використання лапки усередині строки, що відділена такими самими лапками, її потрібно екранувати символом \ попереду. Лапку усередині строки, що відділена іншими лапками, екранувати не потрібно.

Якщо у строці багато символів \, то варто використовувати raw-строки, що подавляють екранування. Для цього перед строкою потрібно написати символ r.

Для завдання багатострокових блоків тексту можна використовувати потрійні лапки.

Приклад

```
s='Hello world'
type(s) #тип
Out[1]: str

s="Hello world"
type(s)
Out[2]: str

s='It\'s string'

s="It's string"

s=r'C:\dir\file.txt'

s='''It's a very big
string'''
s
Out[3]: "It's a very big\nstring"

s1='Hello,'
s2='world'
```

Для конкатенації строк використовується символ +, а для дублювання строки – символ *.

Приклад

```
s1 + s2
Out[1]: 'Hello,world'

s3='!'

s1 + s2 + s3 * 3
Out[2]: 'Hello,world!!!'
```

Довжину строки можна дізнатися за допомогою функції **len**(string).

Приклад

```
s='Hello world'

len(s)
Out[1]: 11
```

a. Індексація

Доступ до символів строки здійснюється за допомогою квадратних дужок та порядкового номеру символу у строці. Цей номер відраховується з нуля – з початку строки. Для отримання символу строки з кінця можна використати від'ємні номери. Так, -1 буде відповідати останньому символу строки.

Приклад

```
s='Hello world'
s[0]
Out[1]: 'H'

s[-1]
Out[2]: 'd'
```

За допомогою квадратних дужок можна вибрати не один, а декілька символів строки, що йдуть підряд. Така конструкція називається slicing (зріз) та має наступну схему:

string[start:stop+step:step]

Тут:

- start – номер першого елементу зрізу;
- stop – номер останнього елементу зрізу;
- stop – крок.

Зауважимо, що конструкція string[start:stop:step] поверне усі елементи з номеру start по номер stop-step з кроком step.

Не всі елементи зрізу є обов'язковими. Так:

- конструкція string[start:stop] еквівалентна конструкції string[start:stop:1];
- конструкція string[start:] еквівалентна конструкції string[start:-1:1];

- конструкція string[:stop] еквівалентна конструкції string[0:stop:1];
- конструкція string[::step] еквівалентна конструкції string[0:-1:step];
- конструкція string[:] або string[::] еквівалентна конструкції string[0:-1:1] – вибірка усіх символів строки.

Приклад

```
s[0:5]
Out[1]: 'Hello'

s[4:5]
Out[2]: 'o'

s[:5]
Out[3]: 'Hello'

s[6:]
Out[4]: 'world'

s[:]
Out[5]: 'Hello world'

s[::1]
Out[6]: 'Hello world'

s[::2]
Out[7]: 'Hlowrd'

s[::-1]
Out[8]: 'dlrow olleH'
```

Строки є незмінюваними послідовностями, тому змінити вже створену строку неможливо, можна лише створити нову змінену строку.

Приклад

```
s[3]='t'
Traceback (most recent call last):
```

```
  File "<ipython-input-40-262439edcb7c>", line 1, in <module>
    s[3]='t'

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

b. Пошук та заміна підстрок

Для пошуку підстроки у Python є наступні функції:

- string.**find**(substring) – повертає індекс першого входження підстроки substring у строку string; якщо підстрока substring не міститься у строці string, повертає -1;
- string.**rfind**(substring) – аналогічно до функції find, але повертає індекс останнього входження підстроки substring у строку string;
- string.**count**(substring) – підраховує кількість входжень підстроки substring у строку string.

Приклад

```
s='Hello world'
x='o'

s.find(x)
Out[1]: 4

s.rfind(x)
Out[2]: 7

s.count(x)
Out[3]: 2

x='or'

s.find(x)
Out[4]: 7

s.rfind(x)
Out[5]: 7

s.count(x)
Out[6]: 1
```

Для заміни підстроки в Python є функція string.**replace**(old, new), що замінює у строці string підстроку old на підстроку new.

Приклад

```
s2=s.replace('world', "test")
s2
Out[1]: 'Hello test'
```

Для перевірок відносно підстрок у Python є наступні функції:

- string.**startswith**(substring) – перевіряє, чи починається строка string з підстроки substring;
- string.**endswith**(substring) – перевіряє, чи закінчується строка string підстрокою substring.

До будь-якої з цих функцій можна подати в якості substring не одну строку, а декілька, помістивши їх у круглі дужки та розділивши комою. У такому разі виконується перевірка, чи починається/закінчується строка string з однієї з підстрок, вказаних у substring.

Приклад

```
s='hello world'

s.startswith('hello')
Out[1]: True

s.startswith(('hello', 'hi'))
Out[2]: True

s.endswith('world')
Out[3]: True
```

Для видалення зайвих пробілів у Python є наступні функції:

- string.**lstrip**() – видаляє пробіли на початку строки;
- string.**rstrip**() – видаляє пробіли в кінці строки;
- string.**strip**() – видаляє пробіли на початку та в кінці строки.

Приклад

```
s='  hello'
s.lstrip()
Out[1]: 'hello'

s='hello '
```

s.**rstrip**()
Out[2]: 'hello'

s=' hello '
s.**strip**()
Out[3]: 'hello'

c. Регістри

Для переводу строки в інші регістри в Python є наступні функції:

- string.**capitalize**() – переводить першу букву в строці у верхній регістр;
- string.**upper**() – переводить строку у верхній регістр;
- string.**lower**() – переводить строку у нижній регістр;
- string.**swapcase**() – змінює регістри букв.

Приклад

s='hello world'
s.**capitalize**()
Out[1]: 'Hello world'

s1=s.**upper**()
s1
Out[2]: 'HELLO WORLD'

s1.**lower**()
Out[3]: 'hello world'

s.**swapcase**()
Out[4]: 'HELLO WORLD'

s1.swapcase()
Out[5]: 'hello world'

Функція **ord**(symbol) повертає код символа, а функція **chr**(code) повертає односимвольну строку, що відповідає заданому коду символа.

Приклад

s='z'
i=**ord**(s)

```
i
Out[1]: 122

chr(i)
Out[2]: 'z'
```

d. Перевірки

Для перевірок відносно змісту строки в Python є наступні функції:

- string.**isdigit**(), string.**isdecimal**(), string.**isnumeric**() – перевіряє, чи є всі символи у строці цифрами;
- string.**isalpha**() – перевіряє, чи є всі символи у строці буквами;
- string.**isalnum**() – перевіряє, чи є всі символи у строці цифрами або буквами.

Приклад

```
s='12345'

s.isdigit()
Out[1]: True

s.isdecimal()
Out[2]: True

s.isnumeric()
Out[3]: True

s.isalpha()
Out[4]: False

s.isalnum()
Out[5]: True

s='hello'

s.isalpha()
Out[6]: True

s.isalnum()
Out[7]: True
```

```
s.isdigit()
Out[8]: False
```

2. Списки

Список – це структура даних, що містить неупорядкований набір елементів, можливо, різнотипних.

Елементи списку записуються у квадратних дужках через кому.

Приклад

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

print(l)
[1, 2, 3, 4]

l
Out[163]: [1, 2, 3, 4]

print(type(l))
<class 'list'>
```

Кількість елементів списку можна визначити за допомогою функції **len**(list).

Приклад

```
len(l)
Out[1]: 4
```

a. Індексація

Доступ до елементів списку здійснюється за допомогою квадратних дужок та порядкового номеру елементу у списку. Цей номер відраховується з нуля – з початку списку. Для отримання елементу списку з кінця можна використати від'ємні номери. Так, -1 буде відповідати останньому елементу списку.

Приклад

```
l=[1, 2, 3, 4]
l[0]
Out[1]: 1

l[-1]
```

Out[2]: 4

За допомогою квадратних дужок можна вибрати не один, а декілька елементів списку, що йдуть підряд. Така конструкція називається slicing (зріз) та має наступну схему:

list[start:stop+step:step] – вибірка елементів зі списку з номера start по номер stop включно з кроком step. Зрізи у списках аналогічні зрізам у строках.

Приклад

```
l[1:3]
Out[1]: [2, 3]

l[::2]
Out[2]: [1, 3]

l=[1, 'a', 1.0, 1-1j]
print(l)
[1, 'a', 1.0, (1-1j)]

l[1:3]=[5,10]
print(l)
[1, 5, 10, (1-1j)]

nested_list=[1, [2, [3, [4, [5]]]]]
nested_list
Out[3]: [1, [2, [3, [4, [5]]]]]
```

Для генерації списку в Python є наступна функція:

range(start, stop+step, step), що генерує список, який містить такі елементи: start, start+step,… stop-step, stop.

Аналогічно до зрізів у функції range можна не вказувати усі параметри.

Функція range повертає не список, а сам генератор, для перетворення його до списку можна скористатися функцією **list**(range(start, stop, step)).

Приклад

```
start=10
stop=30
```

step=2
range(start, stop, step)
Out[1]: range(10, 30, 2)

list(range(start, stop, step))
Out[2]: [10, 12, 14, 16, 18, 20, 22, 24, 26, 28]

b. Сортування

За допомогою функції **list**(arg) можна перетворити строку у список символів.

Функція list.**count**(element) підраховує кількість елементів списку із заданним значенням element.

Приклад

s='Hello world'

l=**list**(s)
l
Out[1]: ['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']

l.**count**('l')
Out[2]: 3

Для сортування елементів списку в алфавітному порядку або в порядку зростання в залежності від їх типу в Python є наступні функції:

- **sorted**(list) – повертає відсортований список, вихідний список не змінюється;
- list.**sort**() – змінює порядок елементів у списку.

Обидві функції мають додатковий параметр **reverse**, який за замовчуванням дорівнює False. Значення reverse=True показує виконувати сортування в оберненому порядку: в порядку спадання елементів чи в порядку, протилежному до алфавітного, в залежності від типу елементів, що містяться у списку.

Приклад

sorted(l)
Out[1]: [' ', 'H', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r', 'w']

l
Out[2]: ['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']

l.**sort**()

```
print(l)
[' ', 'H', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r',
'w']

l.sort(reverse=True)
print(l)
['w', 'r', 'o', 'o', 'l', 'l', 'l', 'e', 'd', 'H',
' ']

l=[5,2,9,0,-3]
l.sort()
print(l)
[-3, 0, 2, 5, 9]
```

c. Додавання та видалення елементів

Пустий список задається пустими квадратними дужками.

Для перевірки того, чи знаходиться елемент у списку, можна скористатися наступною конструкцією: element **in** list – проста перевірка на приналежність елементу element у списку list, повертає True або False;

Функція list.**index**(element) повертає індекс першого входження елементу element у списку list, якщо даний елемент не міститься у списку, генерується виключення.

Зміну значень елементів списку можна проводити за їх порядковими номерами (індексами). Причому можна змінювати 1 елемент або декілька суміжних, використовуючи зрізи.

Для додавання елементу до списку можна скористатися однією з наступних функцій:

- list.**append**(element) – додає елемент element в кінець списку list;
- list.**insert**(index,element) – вставляє елемент element на позицію index у списку list.

Приклад

```
l=[]

l.append('A')
```

```
l.append("d")
l.append('d')

print(l)
['A', 'd', 'd']

'A' in l
Out[1]: True

l.index('A')
Out[2]: 0

l.index('d')
Out[3]: 1

l.index('f')
Traceback (most recent call last):

  File "<ipython-input-7-9f5a19c78694>", line 1, in <module>
    l.index('f')

ValueError: 'f' is not in list

l[1]='p'
l[2]='p'

print(l)
['A', 'p', 'p']

l[1:3]=['d','d']

l
Out[4]: ['A', 'd', 'd']

l.insert(0,'i')
l.insert(1,'n')
l.insert(2,'s')
l.insert(3,'e')
```

```
l.insert(4,'r')
l.insert(5,'t')

print(l)
['i', 'n', 's', 'e', 'r', 't', 'A', 'd', 'd']
```

Для видалення елементів зі списку можна скористатися однією з наступних конструкцій:

- list.**remove**(element) – видаляє елемент element зі списку list – видалення за значенням;
- **del** list[index] – видаляє елемент з індексом index зі списку list – видалення за порядковим номером;
- list.**pop**(index) – видаляє елемент з індексом index зі списку list та повертає його значення – видалення за порядковим номером.

Приклад

```
l.remove("A") #удаление символа "A"
print(l)
['i', 'n', 's', 'e', 'r', 't', 'd', 'd']

l.insert(6, 'A')
l
Out[1]: ['i', 'n', 's', 'e', 'r', 't', 'A', 'd', 'd']

l.remove('d') #удаляет 1 символ с конца списка
print(l)
['i', 'n', 's', 'e', 'r', 't', 'A', 'd']

del l[4] #удаление 4-го элемента списка
del l[3]

print(l)
['i', 'n', 's', 't', 'A', 'd']

l.pop(4) #удаляет 4-ый элемент списка и возвращает его
Out[2]: 'A'
```

l.pop() #если номер элемента не задан, удаляет последний элемент списка и возвращает его значение
Out[3]: 'd'

print(l)
```
['i', 'n', 's', 't']
```

d. Розбиття строки

Функція string.**split**() використовується для розбиття строки string на підстроки з роздільником – пробілом. Якщо для розбиття повинен використовуватися інший роздільник, то його можна вказати як параметр функції split. Ця функція приймає у якості роздільника лише один параметр, для розбиття за більш ніж одним роздільником, потрібно використовувати регулярні вирази.

l2='a b c'.**split**()
print(l2)
```
['a', 'b', 'c']
```

l2='a.b.c'.split('.')
print(l2)
```
['a', 'b', 'c']
```

3. Кортежи

Кортежі служать для збереження декількох об’єктів, можливо, різнотипних, разом.

Елементи кортежу записуються у круглих дужках через кому або навіть без дужок через кому.

Доступ до елементів кортежу здійснюється за допомогою квадратних дужок та порядкового номеру елементу у кортежі – аналогічно доступу до елементів списку. Зрізи також підтримуються.

Кортеж відноситься до незмінних типів даних, тобто не можна змінити якійсь елемент кортежу, можна лише створити новий кортеж.

Приклад

point=(10, 20)

print(point, type(point))
```
(10, 20) <class 'tuple'>
```

```
point=10, 20

print(point,type(point))
(10, 20) <class 'tuple'>

x, y=point
print('x=', x)
x= 10
print('y=', y)
y= 20

point[0]=20
Traceback (most recent call last):

  File "<ipython-input-125-c7be33370c3d>", line 1, in <module>
    point[0]=20

TypeError: 'tuple' object does not support item assignment

myTuple='a', 10, 1j, True
print(myTuple)
('a', 10, 1j, True)

myTuple[1]
Out[0]: 10

myTuple[1:3:2]
Out[1]: (10,)
```

Функція string.**partition**(delimiter) використовується для розбиття строки string на підстроки з роздільником delimiter. На відміну від функції split, partition повертає кортеж, а не список.

Приклад

```
s='First sentence. Second sentence'

s.partition('.')
Out[1]: ('First sentence', '.', ' Second sentence')
```

a. Обмін значеннями

Завдяки наявності кортежів обмін значеннями двох (та більше) змінних можна записати в одну строку.

Приклад

```
a=1; b=2
a, b = b, a
print(a, b)
2 1
```

4. Словники

Словники задаються парами: ключ та значення одним зі способів:

- dictionary=**{key1:value1, key2:value2, …}**;
- dictionary=**dict(key1=value1, key2=value2, …)**.

При цьому ключі повинні бути унікальними, значення можуть повторюватися.

Доступ до елементів словника здійснюється за допомогою квадратних дужок та ключа. Словники відносяться до змінних типів даних, тому за допомогою квадратних дужок та ключа можна змінити значення за даним ключем.

Функція dictionary.**items**() повертає усі елементи словника у вигляді списку кортежів.

Функція dictionary.**keys**() повертає усі ключі словника у вигляді списку.

Функція dictionary.**values**() повертає усі значення, що зберігаються у словнику у вигляді списку.

Для додавання нового елементу потрібно виконати присвоєння за новим ключем. В такому разі нова пара (ключ, значення) додається в кінець словника.

Приклад

```
params={'parameter1':1.0, 'parameter2':2.0, 'parameter3':3.0}

print(type(params))
<class 'dict'>

print(params)
{'parameter1':     1.0,     'parameter2':     2.0,
'parameter3': 3.0}
```

params=**dict**(parameter1=1.0, parameter2=2.0, parameter3=3.0)
print(params)
```
{'parameter1':     1.0,     'parameter2':     2.0,
'parameter3': 3.0}
```

print("parameter1="+str(params['parameter1']))
```
parameter1=1.0
```
print("parameter2="+str(params['parameter2']))
```
parameter2=2.0
```
print("parameter3="+str(params['parameter3']))
```
parameter3=3.0
```

params['parameter1']='A'

params['parameter4']='D'

params
Out[1]: {'parameter1': 'A', 'parameter2': 2.0, 'parameter3': 3.0, 'parameter4': 'D'}

print(params.**items**())
```
dict_items([('parameter1',   'A'),   ('parameter2',
2.0), ('parameter3', 3.0), ('parameter4', 'D')])
```

print(params.**keys**())
```
dict_keys(['parameter1',              'parameter2',
'parameter3', 'parameter4'])
```

print(params.**values**())
```
dict_values(['A', 2.0, 3.0, 'D'])
```

Для видалення елементів зі словника можна скористатися однією з наступних конструкцій:

- dictionary.**pop**(key) – видаляє елемент з ключем key зі словника dictionary та повертає його значення;
- **del** dictionary[key] – видаляє елемент з ключем key зі словника dictionary.

Приклад

```
params={'parameter1':1.0, 'parameter2':2.0, 'parameter3':3.0}

params.pop('parameter2')
Out[1]: 2.0

params
Out[2]: {'parameter1': 1.0, 'parameter3': 3.0}

del params['parameter1']

print(params)
{'parameter3': 3.0}
```

Для перевірки того, чи знаходиться елемент в словнику, можна скористатися наступною конструкцією: key **in** dictionary – перевірки на приналежність елементу з ключем key у словнику dictionary, повертає True або False.

Приклад

```
'parameter3' in params #проверка наличия ключа в словаре
Out[1]: True

3.0 in params #проверка производится по ключу, а не по значенню!
Out[2]: False
```

За допомогою функції dict(**zip**(keys, values)) можна створити словник із двох списків.

Приклад

```
keys=['a','b','c']
vals=[1,5,3,-2]
D=dict(zip(keys, vals)) #таким образом можно создать словарь из 2 списков
D
Out[1]: {'a': 1, 'b': 5, 'c': 3}
```

5. Множини

Множина – неупорядкований набір простих елементів. Множина задається за допомогою функції **set**(consequence), де consequence – це

список, кортеж чи строка, або ж перерахування елементів у фігурних дужках.

У множині не має елементів, що повторюються, порядок вільний.

Приклад

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

len(S)
Out[1]: 3
```

Для виконання теоретико-множинних операцій у Python є наступні функції:

- set1.**intersection**(set2) або set1&set2 – пересічення множин;
- set1.**union**(set2) або set1|set2 – об'єднання множин;
- set1.**difference**(set2) або set1-set2 – різниця множин;
- set1.**symmetric_difference**(set2) або set1^set2 – симетрична різниця множин.

Приклад

```
S2={1,4,2,5,1,4}

S.intersection(S2)
Out[1]: {1, 2}

S.union(S2)
Out[2]: {1, 2, 3, 4, 5}

S2.union(S)
Out[3]: {1, 2, 3, 4, 5}

S.difference(S2)
Out[4]: {3}

S2.difference(S)
Out[5]: {4, 5}

S.symmetric_difference(S2)
Out[6]: {3, 4, 5}
```

Для додавання елементу до множини використовується функція set.**add**(element) – додає елемент element до множини set.

Для видалення елементу із множини використовується функція set.**remove**(element) – видаляє елемент element із множини set.

Приклад

```
S.add(8)
print(S)
{8, 1, 2, 3}

S.remove(1)
print(S)
{8, 2, 3}
```

Ще один варіант завдання множини - **frozenset**(consequence) – створює незмінну множину. Основна від'ємність від звичайної множини – до frozenset неможливо додати чи видалити елемент.

Приклад

```
F=frozenset([1,2,4,3,1,2])

print(F)
frozenset({1, 2, 3, 4})

F.add(5) #добавление или удаление элемента в frozenset невозможно
Traceback (most recent call last):

  File "<ipython-input-22-d713ddb2718c>", line 1, in <module>
    F.add(5)

AttributeError: 'frozenset' object has no attribute 'add'
```

6. Копіювання

При простому присвоєнні змінної типу списку копіювання списку не відбувається, копіюються лише посилання. Тому при зміні елементів списку за новою змінною відбудеться зміна списку за старою змінною та навпаки.

Приклад

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

```
B=A

A[0]=5
print(A)
[5, 2, 3]

print(B)
[5, 2, 3]
```

Щоб створити копію списку можна скористатися функцією list.**copy**(), що створює копію списку та повертає її. У такому разі зміна елементів одного списку ніяк не вплине на елементи другого списку.

Приклад

```
C=A.copy()

A[0]=1
print(A)
[1, 2, 3]

print(C)
[5, 2, 3]
```

Ще один спосіб створити копію – використати зрізи. Вираз list[::] поверне копію усього списку, яку можна присвоїти новій змінній.

Приклад

```
D=A[::]

A[0]=10
print(A)
[10, 2, 3]

print(D)
[1, 2, 3]
```

Аналогічно для словників.

Приклад

```
A={1:10,2:20}
B=A
```

```
A[1]=100
A
Out[1]: {1: 100, 2: 20}

B
Out[2]: {1: 100, 2: 20}

C=A.copy()

A[2]=200
A
Out[3]: {1: 100, 2: 200}

C
Out[4]: {1: 100, 2: 20}
```

7. Завдання та вправи

1. Дана строка.
   1) Вивести третій символ строки.
   2) Вивести передостанній символ строки.
   3) Вивести перші п’ять символів строки.
   4) Вивести усю строку, окрім останніх двох символів.
   5) Вивести усі символи з парними індексами (враховуючи, що індексація починається з нуля).
   6) Вивести усі символи з непарними індексами.
   7) Вивести усі символи в оберненому порядку.
   8) Вивести усі символи строки через один в оберненому порядку, починаючи з останнього.
   9) Вивести довжину строки.
2. Дана строка, в якій буква h зустрічається як мінімум два рази. Потрібно видалити з цієї строки перше та останнє входження букви h, а також усі символи, що знаходяться між ними.
3. Дан список чисел. Визначити, скільки в ньому зустрічається різних чисел. Цю задачу в Python можна розв’язати в одну строку.
4. Дани два списки чисел. Підрахувати, скільки чисел міститься одночасно як у першому списку, так і у другому. Цю задачу в Python можна розв’язати в одну строку.

V Ввід, вивід даних

1. Ввід даних

Для вводу даних у Python є функція **input**(). Вона повертає введені в строці консолі символи строкового типу. Дана функція зчитує усю строку. В якості параметра функції **input**(string) можна використовувати строку-підказку string для користувача.

Приклад

```
x=input()
2

x
Out[1]: '2'

s=input('Input string: ')
Input string: my string

s
Out[2]: 'my string'
```

Якщо вводяться дані не строкового, а, наприклад, цілого типу, їх можна отримати наступним чином: **int**(input()), тобто введена строка одразу перетворюється до цілого типу.

Приклад

```
x=int(input())
5

x
Out[2]: 5

x=int(input('Input digit: '))
Input digit: 5

x
Out[10]: 5
```

2. Вивід даних

Для вводу даних у Python є функція **print**(), що має наступну сигнатуру:

print(value,… sep=' ', end='\n', file=sys.stdout, flush=False)
Тут:

- **value** – те/ті значення, що потрібно вивести, якщо значень декілька; вони розділяються комою;
- **sep** – роздільник між різними значеннями при виводі, за умовчуванням – пробіл;
- **end** – символ, що ставиться після виводу останнього значення, за умовчуванням символ нової строки;
- **file** – файл, куди проводиться вивід, за умовчуванням sys.stdout;
- **flush** – примушувати чи ні вивід до потоку.

a. Стандартна конкатенація

При виводі значень декількох змінних їх можна задати через кому, причому ці змінні можуть мати різні типи. Строкові змінні можна конкатенувати в одну строку, тоді між ними не буде роздільників.

Приклад

```
print("s1", 's2', 's3')
s1 s2 s3

print("s1", 's2', 's3', sep=';')
s1;s2;s3

print("s1", 's2', 's3', sep=';', end='.')
s1;s2;s3.

print('s1', 1.0, False, -1j)
s1 1.0 False (-0-1j)

print("s1"+'s2'+'s3')
s1s2s3
```

b. С-образний вивід

Для комбінації виводу строкових та числових змінних можна використовувати так званий С-образний вивід. Цілі числа позначаються як %d, дійсні – як %f, причому для дійсних чисел можна вказувати кількість цифр після коми: %.nf, де n – ціле число.

Приклад

```
print('value=%f' % 1.0)
```

```
value=1.000000
```

```
s2='value1=%.2f, value2=%d' % (3.1415, 1.5)
print(s2)
value1=3.14, value2=1
```

c. Функція format

Для формування строки, що містить дані інших типів, можна використовувати функцію string.**format**(object1, object2,…). Строка string повинна містити фігурні дужки, у яких можуть знаходиться числа 0, 1, 2,… Вони позначають позиції, куди потрібно вставити дані object1, object2,… Функція format підставляє замість аргументів у фігурних дужках свої параметри у тому порядку, як вони вказані. Цифри у дужках не є обов'язковими.

Приклад

```
s3='value1={0}, value2={1}'.format(3.1415, 1.5)
print(s3)
value1=3.1415, value2=1.5
```

```
s3='value1={1}, value2={0}'.format(3.1415, 1.5)
print(s3)
value1=1.5, value2=3.1415
```

```
s3='value1={}, value2={}'.format(3.1415, 1.5)
print(s3)
value1=3.1415, value2=1.5
```

Можлива робота зі списками або кортежами, у цьому випадку використовується подвійна нумерація.

Приклад

```
l=['Masha','Petya']

print('{0[0]} and {0[1]} are sitting at the same desk'.format(l))
Masha and Petya are sitting at the same desk
```

За допомогою функції format можна відобразити числа у різних системах числення.

Приклад

```
print('int: {0: d}, bin: {0: b}, hex: {0: x}, oct: {0: o}'.format(51))
int: 51, bin: 110011, hex: 33, oct: 63
```

За допомогою функції format можна вирівнювати тест:

- < - вирівнювання зліва
- > - вирівнювання справа
- ^ - вирівнювання по центру

Вирівнювання можна проводити із заповнювачем або без нього.

Приклад

```
print('{:<30}'.format('left aligned'))
left aligned

print('{:>30}'.format('right aligned'))
                  right aligned

print('{:^30}'.format('centered'))
             centered

print('{:*^30}'.format('centered'))
**********centered**********
```

Функція format також дозволяє регулювати кількість знаків після коми при виводі.

Приклад

```
print('{:.3f}'.format(1/3))
0.333

points=26
total=30
print('Correct answers: {:.2%}'.format(points/total)) #
Correct answers: 86.67%
```

d. Форматовані строки

Починаючи з версії Python 3.6 для виводу даних також можуть бути використані форматовані строки, які починаються з префіксу **f**. Синтаксис форматованих строк подібний до синтаксису функції format.

Приклад

```
x=3.1415; y=1.5
s=f'value1={x}, value2={y}'
print(s)
value1=3.1415, value2=1.5

print(f'value={1/3:1.3f}')
value=0.333

print(f'value={1/3:1.2%}')
value=33.33%

x=12
print(f'int: {x: d}, bin: {x: b}, hex: {x: x}, oct: {x: o}')
int:  12, bin:  1100, hex:  c, oct:  14
```

e. Функція join для списків строк

Функція delimiter.**join**(list) конкатенує елементи списку (або кортежу) list в одну строку з роздільником delimiter. При цьому елементи списку list повинні бути строкового типу.

Приклад

```
l=['a', 'b', 'c']
print(' '.join(l))
a b c
```

3. Завдання та вправи

1. n школярів ділять k яблук порівну, залишок, що не ділиться, залишається у кошику. Скільки яблук дістанеться кожному школяреві? Скільки яблук залишиться у кошику? Програма отримує на вхід числа n і k і повинна вивести шукану кількість яблук (два числа).
2. Ввести 2 імені та вивести строку формата «<ім’я1> та <ім’я2> друзі» мінімум 4 різними способами.

VI Умовні оператори, цикли

1. Блоки

В умовних операторах та циклах, а також функціях і класах, присутні **відступи**, які замінюють фігурні дужки C++ та позначать окремі блоки. В одному блоці відступи повинні бути однаковими. У

якості відступів можуть виступати знаки табуляції або декілька пробілів. Spyder автоматично пропонує відступи після умовних операторів, циклів, функцій та класів.

Якщо відступи розставлені невірно, програма може працювати некоректно або взагалі не запуститися.

2. Умовний оператор

Умовний оператор має наступну схему:

```
if condition1:
   do_something1
elif condition2:
   do_something2
…
else:
   do_something_else
```

Перевіряється умова condition1. Якщо вона має місце (True), то виконуються дії do_something1, після завершення яких виконуються оператори, що стоять після умовного оператору. Якщо умова condition1 не має місця (False), перевіряється наступна умова condition2, і так далі. Якщо жодна з умов не має місця, виконуються дії do_something_else.

Приклад

```
statement1=False

if statement1:
   print('statement1 is True')
else:
   print('statement1 is False')
```

```
statement1 is False
```

В умовному операторі можу бути відсутня гілка else та гілки elif.

Приклад

```
if statement1:
   print('statement1 is True')
```

Умовний оператор без гілок elif та з гілкою else можна записати в одну строку за наступною схемою:

```
do_something if condition else do_something_else
```

При цьому дії do_something виконуються, якщо умова condition має місце, а дії do_something_else – якщо умова condition не має місця.

Приклад

```
statement1=False
print('statement1 is True') if statement1 else print('statement1 is False')
statement1 is False
```

Оператора множинного вибору switch у Python немає, замість нього можна використовувати if з декількома гілками elif.

Приклад

```
statement2=False

if statement1:
   print('statement1 is True')
elif statement2:
   print('statement2 is True')
else:
   print('statement1 and statement2 are False')
statement1 and statement2 are False
```

Якщо умова складається з декількох виразів, що з'єднані союзом АБО, то наступна умова не перевіряється, якщо не виконана попередня.

Приклад

```
a=5
x='hi'
if a==5 or int(x)==10:
       print('OK')
OK
```

3. Цикл for

Цикл for має наступну структуру:

```
for item in sequence:
   do_something
```

Змінна item послідовно приймає усі значення зі списку/кортежу/строки sequence. У випадку, коли змінна item повинна прийняти усі значення з певного діапазону, зручно використовувати функцію range() у якості sequence.

Приклад

```
for x in [1,2,3]:
   print(x)
1
2
3

for x in range(4):
   print(x)
0
1
2
3

for x in range(-3, 3):
   print(x)
-3
-2
-1
0
1
2

for x in range(-2, 5, 2):
   print(x)
-2
0
2
4

for word in ['scientific', 'computing', 'with', 'Python']:
   print(word)
scientific
computing
with
Python
```

Для проходження по словнику, зручно користоватися функцією items(), яка повертає ключі та відповідні значення словника.

Приклад

```
params={'parameter1':1.0, 'parameter2':2.0, 'parameter3':3.0}
for key, value in params.items():
   print(key+"="+str(value))
parameter1=1.0
parameter2=2.0
parameter3=3.0
```

У випадку, коли потрібні як самі елементи sequence, так і їх порядкові номера, можна скористатися функцією **enumerate**(sequence). За замовчуванням номера починаються у нуля. Якщо початковому елементу з sequence потрібно присвоїти інший номер, його можна вказати другим аргументом функції enumerate(sequence, start_number).

Приклад

```
for idx, x in enumerate(range(-3,3)):
   print(idx, x)
0 -3
1 -2
2 -1
3 0
4 1
5 2

for idx, x in enumerate(range(-3,3), 1):
   print(idx, x)
1 -3
2 -2
3 -1
4 0
5 1
6 2
```

Для синхронного проходження по 2 спискам/кортежам/строкам, що мають однакову довжину, можна скористатися функцією **zip**(sequence1, sequence2).

Приклад

```
l1=[1,2,3]
```

```
l2=[5,6,7]
for x, y in zip(l1, l2):
   print(x, y)
1 5
2 6
3 7
```

4. Генератор списку

Якщо у циклу потрібно обчислити елементи списку, це можна зробити за допомогою генератору списку, що має наступну структуру:
NewList=**[**calculate_element **for** element **in** OldList**]**

Приклад

```
l1=[x**2 for x in range(0,5)]
print(l1)
[0, 1, 4, 9, 16]
```

Усередині генератору списку можна перевіряти умову за наступною схемою:
NewList=**[**calculate_element **for** element **in** OldList **if** condition**]**

Приклад

```
l=[x**2 for x in range(0,5) if x!=2]
print(l)
[0, 1, 9, 16]
```

Можливі вкладені генератори списку з перевіркою умови або без неї.

Приклад

```
l=[x+y for x in 'Hello' if x!='l' for y in 'world' if y!='o']
print(l)
['Hw', 'Hr', 'Hl', 'Hd', 'ew', 'er', 'el', 'ed',
'ow', 'or', 'ol', 'od']
```

Генератор списку можна використовувати, наприклад, для переведення елементів списку до строкового типу.

Приклад

```
print(' '.join(l1))
Traceback (most recent call last):
```

```
  File "<ipython-input-168-b0dac369f913>", line 1, in <module>
    print(' '.join(l1))

TypeError: sequence item 0: expected str instance, int found
```

```
l1=[str(x) for x in l1]
print(l1)
['0', '1', '4', '9', '16']

print(' '.join(l1))
0 1 4 9 16
```

Генератори списків можна використовувати для створення та заповнення двовимірних або багатовимірних списків.

Приклад

```
n=5
A=[[0]*n for i in range(n)]
print(A)
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

A=[[x*y for x in range(5)] for y in range(3)]
print(A)
[[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8]]
```

Окрім генератора списку в Python існують генератори словника та генератори множини, що мають аналогічні структури.

Приклад

```
d={x: x**2 for x in range(5)}
print(d)
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

s={x**2 for x in range(5)}
print(s)
{0, 1, 4, 9, 16}
```

5. Цикл while

Цикл while, що виконується, поки вірна деяка умова, має наступну структуру:

```
while condition:
   do_something
```

Приклад

```
i=0
while i<5:
   print(i)
   i+=1
0
1
2
3
4
```

6. Оператор break

Для переривання циклу while або циклу for можна використовувати оператор **break**. При його наявності у циклу while або циклу for відповідно може бути присутня гілка else, яка буде виконана одразу після циклу, якщо цикл не був перерваний оператором break.

Приклад

```
x=3

while x<5:
   x+=2
   if x==5:
      print('Break: x=5')
      break
else: print('x=', x)
Break: x=5

x=2

while x<5:
   x+=2
   if x==5:
      print('Break: x=5')
```

```
        break
else: print('x=', x)
x= 6

for x in range(3,9):
    if x==5:
        print('Break: x=5')
        break
else: print('x=',x)
Break: x=5

for x in range(3,9,3):
    if x==5:
        print('Break: x=5')
        break
else: print('x=',x)
x= 6
```

7. Оператор continue

Для переходу на нову ітерацію циклу while або циклу for можна використовувати оператор **continue**.

Приклад

```
x=3

while x<9:
    x+=2
    if x==5:
        continue
    print(x)
7
9

for x in range(3, 9, 2):
    if x==5:
        continue
    print(x)
3
7
```

8. Завдання та вправи

1. В математиці функція sign(x) (знак числа) визначена так:
   sign(x) = 1, якщо x > 0,
   sign(x) = -1, якщо x < 0,
   sign(x) = 0, якщо x = 0.
   Для даного числа x вивести значення sign(x).
2. Дано натуральне число. Потрібно визначити, чи є рік з даним номером високосним. Якщо рік є високосним, вивести YES, інакше вивести NO. У відповідності до григоріанського календаря, рік є високосним, якщо його номер кратний 4, але не кратний 100, а також якщо він кратний 400.
3. Дано два цілих числа A і B (при цьому A ≤ B). Потрібно вивести усі числа від A до B включно.
4. Дано два цілих числа A і B. Потрібно вивести усі числа від A до B включно у порядку зростання, якщо A< B, або в порядку спадання в противному випадку.
5. Програма отримує на вхід послідовність цілих невід'ємних чисел, кожне число записано в окремій строці. Послідовність закінчується числом 0, при зчитуванні якого програма повинна закінчити свою роботу і вивести кількість членів послідовності (не враховуючи число 0). Числа, що слідують за числом 0, не потрібно зчитувати.

VII Функції, виключення

1. Іменовані функції

Іменована функція має наступну структуру:

```
def func(parameters):
  #do_something
  return value
```

Тут:

- **func** – назва функції;
- **parameters** – параметри функції, що передаються через кому; функція може не мати вхідних параметрів;
- **value** – вихідний параметр/параметри; функція може не мати вихідних параметрів.

Завдяки наявності типу кортеж можливо повертати декілька параметрів, що записуються через кому. Якщо у функції відсутні вихідні параметри, return не обов'язковий.

Приклад

```
def powers(x):
    return x**2, x**3, x**4

powers(3)
Out[1]: (9, 27, 81)

x2, x3, x4=powers(3)
print(x3)
27

def f(x,y):
    print('x=',x,'y=',y)

f(5,2)
x= 5 y= 2
```

а. Параметри за замовчуванням

У функції можливі параметри за замовчуванням, які повинні стояти в кінці списку вхідних параметрів. При визові функції з параметрами за замовчуванням, їх можна передавати або у строгому порядку без вказання імен параметрів, або у будь-якому порядку, вказуючи їх імена. Завдяки вказуванню імен можна не вказувати деякі аргументи за замовчуванням, причому вони не обов'язково повинні стояти у кінці списку вхідних аргументів.

Приклад

```
def myfunc(x, p=2, debug=False):
    if debug:
        print("myfunc for x="+str(x)+" using exponent p="+str(p))
    return x**p

myfunc(5)
Out[1]: 25

myfunc(5, debug=True)
myfunc for x=5 using exponent p=2
Out[2]: 25
```

```
myfunc(p=3, debug=True, x=7)
myfunc for x=7 using exponent p=3
Out[3]: 343
```

b. Змінна кількість параметрів

Якщо потрібно визначити функцію, що приймає будь-яку кількість параметрів, то для цього використовується символ * перед іменем параметра:

- *numbers – усі позиційні аргументи з даної позиції і до кінця збираються у кортеж з іменем numbers,
- **dictionary – усі позиційні аргументи з даної позиції і до кінця збираються у словник з іменем dictionary.

Приклад

```
def f(x, *numbers, **dictionary):
    print('x=',x)

    print('numbers:')
    for i in numbers:
        print(i)

    print('dictionary:')
    for i in dictionary:
        print(dictionary[i])

f(5, 1,2,3, a=2,b=6,c=8)
```

```
x= 5
numbers:
1
2
3
dictionary:
2
6
8
```

** можна використовувати і для того, щоб розпакувати зміст словника при передачі аргументів функції; ключі словника і назви аргументів функції повинні співпадати.

Приклад

```
def f(a, b):
    return a+b
D={'a':1, 'b':2}
f(**D)
Out[1]: 3
```

c. Позиційні аргументи

Якщо потрібно задати у функції строго позиційні аргументи (які не можна буде передати у будь-якому порядку зі вказанням їх імен), то вони повинні йти на початку аргументів функції, а після них повинен йти символ /. Аргументи, які будуть йти після /, можуть бути задані як значенням у строгому порядку, так і по їх іменам у довільному (після строго позиційних аргументів) порядку.

Якщо потрібно задати у функції аргументи, що повинні бути передані тільки по їх назвам, то перед ними потрібно поставити символ *.

Приклад

```
def func(pos1, pos2, /, arg1, arg2, *, key1, key2):
    print('Strictly position arguments:', pos1, pos2)
    print('Usual arguments:', arg1, arg2)
    print('Strickly key arguments:', key1, key2)

func(1,2, 3,arg2=4, key1=5,key2=6)
Strictly position arguments: 1 2
Usual arguments: 3 4
Strickly key arguments: 5 6

func(1,2, 3,4, key1=5,key2=6)
Strictly position arguments: 1 2
Usual arguments: 3 4
Strickly key arguments: 5 6

func(1,2, arg1=3,arg2=4, key2=6,key1=5)
Strictly position arguments: 1 2
Usual arguments: 3 4
```

```
Strickly key arguments: 5 6
```

c. Анотації типів

Для покращення читаємості можливо використовувати анотації типів вхідних даних, а також вказувати тип даних, що повертаються:
def func(param1**:type1**,…)**->return_type**
…

Однак навіть у випадку з анотованими змінними можна подати на вхід функції змінні інших типів.

Приклад

```
def func(L:list, s:str, a:float, n:int)->bool:
    if (a in L) or (n in L):
        return True
    print(s)
    return False

func([1,2,3], "not found", 2.5, 5)
not found
Out[1]: False

func('hello, world', 111, 's', 'hi')
111
Out[2]: False
```

Анотування типів можливо навіть для змінних, що мають значення за замовчуванням.

Приклад

```
def f2(L:list, a:int=5)->int:
    return L.count(a)

f2([1,10,3,1,4,5],1)
Out[1]: 2
```

d. Документування функцій

Текст, що вказаний у коментарях після об’явлення функції, служить її документацією. Доступ до цієї документації можна отримати як function**.__doc__** або **help**(function), де function – назва документованої функції.

```
def powers(x):
    '''
    Returns 2nd, 3rd and 4th powers of the digit.
    '''
    return x**2, x**3, x**4

powers.__doc__
Out[1]: '\n    Returns 2nd, 3rd and 4th powers of the digit.\n    '

help(powers)
Help on function powers in module __main__:

powers(x)
    Returns 2nd, 3rd and 4th powers of the digit.
```

2. Неіменовані функції (lambda-функції)

Lambda-функції мають наступну структуру:

lambda param1,…**:** return_param1,…

Вони можуть мати декілька вхідних та вихідних аргументів. У lambda-функціях можливі параметри за замовчуванням та строго позиційні аргументи, але анотації типів недопустимі.

Приклад

```
f1=lambda x: x**2

f1(2)
Out[1]: 4

f=lambda x, y: x*y

f(1,2)
Out[2]: 2

f=lambda x, y=5: x*y

f(2)
Out[3]: 10

f=lambda a,/,b,*,c=3:a+b+c
```

```
f(1,2)
Out[4]: 6
```

Lambda-функцію (так само як і звичайну іменовану функцію) можна указати в якості ключа сортування списка.

Приклад

```
l2=[[11,5],[-3,8],[6,0],[1,4]]

l2.sort(key=lambda x: x[0]**2+x[1]**2)
l2
Out[1]: [[1, 4], [6, 0], [-3, 8], [11, 5]]

sorted(l2, key=lambda x: x[0]**2+x[1]**2)
Out[2]: [[1, 4], [6, 0], [-3, 8], [11, 5]]
```

3. Застосування функції до списка

Іменовану чи неіменовану функцію можна застосувати до списку за допомогою функції **map**().

Приклад

```
map(lambda x: x**2, range(-3,4))
Out[1]: <map at 0x8e9f890>

list(map(lambda x: x**2, range(-3,4)))
Out[2]: [9, 4, 1, 0, 1, 4, 9]

def f(x):
   return x**2

list(map(f, [2,-1,5]))
Out[3]: [4, 1, 25]
```

4. Стандартні функції

Серед стандартних Python-функції варто відмітити наступні:

- **min** – знаходить мінімальне серед заданих чисел або чисел списку;
- **max** – знаходить максимальне серед заданих чисел або чисел списку;
- **sum** – знаходить суму елементів списку.

Приклад

min(5,-1,3)
Out[1]: -1

max(5,-1,3)
Out[2]: 5

l=[2,-5,3,8,0]
min(l)
Out[3]: -5

max(l)
Out[4]: 8

sum(l)
Out[5]: 8

5. Локальні змінні

Усередині блоків можна об'являти **локальні змінні**, область видимості яких буде обмежена блоком, в якому вони об'явлені, починаючи з точки об'явлення імені.

У наступному прикладі виводиться значення локальної змінної, яка була передана в якості параметру. Усередині функціїї змінюється значення локальної змінної, значення глобальної змінної не змінилось.

Приклад

```
x=5

def f(x):
    print('parameter x=', x)
    x=2
    print('local x=', x)

f(x)
print('global x=', x)
parameter x= 5
local x= 2
global x= 5
```

У даному прикладі параметром функції виступає список. Так як списки передаються за посиланнями, зміна значення списку усередині функціїї змінює глобальну змінну.

Приклад

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

def f(l):
    print('l1:', l)
    l[0]=10
    print('l2:', l)

f(l)
print('l3:', l)
```

```
l1: [1, 2, 3]
l2: [10, 2, 3]
l3: [10, 2, 3]
```

У цьому прикладі усередині функції створюється нова змінна, яка за назвою співпадає з параметром функції. Але нова локальна змінна ніяк не впливає на глобальну змінну з таким самим ім'ям.

Приклад

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

def f(l):
    print('l1:',l)
    l=[10,20,30]
    print('l2:',l)
f(l)
print('l3:',l)
```

```
l1: [1, 2, 3]
l2: [10, 20, 30]
l3: [1, 2, 3]
```

Змінні, що об'явлені усередині циклів та умовних операторів, не є локальними.

b. global

Для зміни значення глобальної змінної усередині функції використовуються ключове слово global. Global змінна не може виступати параметром функції.

При використанні ключового слова global усередині функції можна змінювати значення глобальної змінної.

Приклад

```
x=5

def f():
   global x
   print('global x=', x)
   x=2
   print('new global x=', x)
f()
print('the final global x=', x)
```

```
global x= 5
new global x= 2
the final global x= 2
```

c. nonlocal

У випадку вкладених функцій можна використовувати ключове слово nonlocal, яке позначає, що вказана змінна не є локальною відносно внутрішньої функціїї, а є локальною змінною зовнішньої функції. Така змінна також не передається як параметр функції. Зміна її значення усередині внутрішньої функціїї впливає на її значення у зовнішній функції.

Приклад

```
def f_outer():
   x=5

   def f_inner():
      nonlocal x
      x=2
      print('inner x=', x)

   f_inner()
```

```
    print('outer x=', x)

f_outer()
```

```
inner x= 2
outer x= 2
```

Без ключового слова nonlocal значення змінної не змінюється, а у внутрішній функції об'являється нова змінна.

Приклад

```
def f_outer():
    x=5

    def f_inner():
        x=2
        print('inner x=', x)

    f_inner()
    print('outer x=', x)

f_outer()
```

```
inner x= 2
outer x= 5
```

5. Декоратори

Декоратори – це, по суті, «обгортки», які дають можливість змінити поведінку функції, не змінюючи її код. Для цього створюється функція-декоратор, що приймає у якості параметру функцію. Усередині функції-декоратора описується внутрішня функція-«обгортка», що може виконувати деякі дії до та після визову функції-параметру. Функція-декоратор повертає функцію-«обгортку». Власне сам декоратор – це функція-декоратор від функції, яку потрібно продекорувати.

Приклад

```
def my_decorator(f):

    def wrapper_func():
```

```
        print('Do something before function f')
        f()
        print('Do something after function f')

    return wrapper_func

def f():
    print('Function f')

f()
Function f

decorated_f=my_decorator(f)

decorated_f()
Do something before function f
Function f
Do something after function f
```

Якщо функція, яка декорується, у недекорованому вигляді не потрібна, можна записати декоратор іншим чином. Тоді виклик функції буде завжди супроводжуватися викликом функції-декоратора.

Приклад

```
#другой вариант создания декоратора
@my_decorator
def f():
    print('Function f')

f()
Do something before function f
Function f
Do something after function f
```

а. Декілька декораторів

До функції можна застосовувати не тільки один, а й декілька декораторів. При цьому важливо слідкувати за порядком декорування, так як перестановка декораторів не комутативна.

Приклад

```
def my_new_decorator(f):
    def wrapper():
        print('Start new decorator')
        f()
        print('Stop new decorator')
    return wrapper

@my_decorator
@my_new_decorator
def f1():
    print('Function f1')

@my_new_decorator
@my_decorator
def f2():
    print('Function f2')

f1()
print()
f2()
```

```
Do something before function f
Start new decorator
Function f1
Stop new decorator
Do something after function f

Start new decorator
Do something before function f
Function f2
Do something after function f
Stop new decorator
```

b. Декорування функцій з вхідними аргументами

При декоруванні функцій, що мають вхідні аргументи, потрібно слідкувати за тим, щоб функція-«обгортка» мала таку саму кількість вхідних аргументів. Назви аргументів можуть бути будь-якими.

Приклад

```
def decorator(f):
    def wrapper(arg1, arg2):
        print('Wrapper function with arguments:', arg1, arg2)
        f(arg1, arg2)
    return wrapper

@decorator
def f(a, b):
    print('a={}, b={}'.format(a, b))

f(5,8)
```

```
Wrapper function with arguments: 5 8
a=5, b=8
```

Якщо кількість вхідних аргументів невідома, або потрібно створити універсальний декоратор, що може бути використаний для будь-якої функції, то у якості аргументів функції-«обгортки» можна вказати наступні: ***args, ****kwargs**.

Приклад

```
def decorator(f):
    def wrapper(*args, **kwargs):
        print('Wrapper function with arguments:', args, kwargs)
        f(*args, **kwargs)
    return wrapper

@decorator
def f1(a, b):
    print('a={}, b={}'.format(a, b))

@decorator
def f2(a, b, c):
    print(a, b, c)

f1(5,8)
f2(1, 5,12)
```

```
Wrapper function with arguments: (5, 8) {}
a=5, b=8
Wrapper function with arguments: (1, 5, 12) {}
1 5 12
```

6. Виключення

Створення виключної ситуації має наступну схему:

raise Exception(“description of the error”)

Для обробки виключної ситуації використовуються наступні оператори:

```
try:
    #normal code
except:
    #code for error handling
```

Код, що знаходиться усередині блоку **try**, буде виконано у випадку, якщо виключна ситуація не була згенерована. Код, що знаходиться усередині блоку **except**, буде виконано у випадку, якщо виключна ситуація була згенерована. Причому гілка except може мати наступний вигляд:

except Exception as e:

У такому разі виключення має псевдонім **e**. Псевдонім не обов’язково присвоювати.

Приклад

```
def Real_sqrt(x):
   if x>0:
      return x**(1/2)
   else:
      raise Exception('Real_sqrt: result can not be converted to real digit')

x=-5
try:
   print(Real_sqrt(x))
except Exception as e:
   print('Caught an exception: ' + str(e))
```

```
Caught an exception: Real_sqrt: result can not be converted to real digit
```

Гілок except може бути декілька, при цьому кожній гілці буде відповідати вказана виключна ситуація. Також можна описати гілку

else:, що містить код, який буде виконуватися після блоку try-except у випадку, якщо виключних ситуацій не було сгенеровано. У кінці блоку, після else, може бути ще одна гілка **finally:**, що містить код, який буде виконуватися після блоку try-except (чи try-except-else) у будь-якому випадку.

Приклад

```
f=open('text.txt')
l=[]
try:
    for line in f:
        l.append(int(line))
except ValueError:
    print('It is not digit!')
except Exception:
    print('Some other exception')
else:
    print('No exceptions. Everything OK')
finally:
    f.close()
    print('The file is closed')
```

```
It is not digit!
The file is closed
```

Приведемо список можливих виключень:

- **BaseException –** базове виключення, від якого беруть свій початок усі інші.
- **SystemExit** - виключення, що породжується функцією sys.exit при виході з програми.
- **KeyboardInterrupt** – порождується при перериванні програми користувачем (зазвичай сполученням клавиш Ctrl+C).
- **GeneratorExit** – порождується при виклиці методу close об’єкту generator.
- **Exception** – тут закінчуються повністю системні виключення та починаються звичайні, з якими можна працювати.
  - **StopIteration** – порождується вбудованою функцією next, якщо у ітераторі не має більше елементів.
  - **ArithmeticError** – арифметична помилка.

        - **FloatingPointError** – породжується при невдалому виконанні операцій з плаваючою точкою. На практиці зустрічається нечасто.
        - **OverflowError** – виникає, коли результат арифметичної операції занадто великий для уявлення. Не виникає при звичайній роботі з цілими числами (так як python підтримує довгі числа), але може виникнути у деяких інших випадках.
        - **ZeroDivisionError** – ділення на ноль.
    - **AssertionError** – вираз у функції assert хибний.
    - **AttributeError** – об'єкт не має даного атрибуту (значення чи методу).
    - **BufferError** – операція, що пов'язана з буфером, не може бути виконана.
    - **EOFError** – функція натикнулася на кінець файла та не змогла прочитате те, що хотіла.
    - **ImportError** – не вдалось імпортування модуля чи його атрибуту.
    - **LookupError** – некорректний індекс або ключ.
        - **IndexError** – індекс не входить в діапазон елементів.
        - **KeyError** – неіснуючий ключ (у словнику, множині чи іншому об'єкті).
    - **MemoryError** – недостатньо пам'яті.
    - **NameError** – не знайдено змінної з таким іменем.
        - **UnboundLocalError** – зроблено посилання на локальну змінну у функції, але змінна не визначена раніше.
    - **OSError** – помилка, що пов'язана з системою.
        - **BlockingIOError**
        - **ChildProcessError** – невдача при операції з дочірнім процесом.
        - **ConnectionError** – базовий клас для виключень, що пов'язані з підключеннями.
            - **BrokenPipeError**
            - **ConnectionAbortedError**
            - **ConnectionRefusedError**
            - **ConnectionResetError**
        - **FileExistsError** – спроба створити файл або директорію, яка вже існує.
        - **FileNotFoundError** – файл або директорія не існує.

        - **InterruptedError** – системний виклик перерван вхідним сигналом.
        - **IsADirectoryError** – очікувався файл, але це директорія.
        - **NotADirectoryError** – очікувалась директорія, але це файл.
        - **PermissionError** – не вистачає прав доступу.
        - **ProcessLookupError** – вказаного процесу не існує.
        - **TimeoutError** – закінчився час очікування.
    - **ReferenceError** – спроба доступу до атрибуту зі слабким посиланням.
    - **RuntimeError** – виникає, коли виключення не потрапляє ні в одну з інших категорій.
    - **NotImplementedError** – виникає, коли абстрактні методи класу потребують перевизначення у дочірніх класах.
    - **SyntaxError** – синтаксична помилка.
        - **IndentationError** – невірні відступи.
            - **TabError** – змішування у відступах табуляції та пробілів.
    - **SystemError** – внутрішня помилка.
    - **TypeError** – операція застосована до об'єкту невідповідного типу.
    - **ValueError** – функція отримує аргумент правильного типу, але некоректного значення.
    - **UnicodeError** – помилка, що пов'язана з кодуванням / декодуванням unicode у строках.
        - **UnicodeEncodeError** – виключення, що пов'язане з кодуванням unicode.
        - **UnicodeDecodeError** – виключення, що пов'язане з декодуванням unicode.
        - **UnicodeTranslateError** – виключення, що пов'язане з перекладом unicode.
    - **Warning** – попередження.

7. Перевірка assert

Функція assert перевіряє, чи є вираз істинним чи хибним. Якщо вираз істинний, то виконується наступна строка коду і так далі. Якщо вираз хибний, то генерується виключення AssertionError.

Приклад 1

```
def func(a,b):
    assert b!=0
```

```
    return a/b

func(2,5)
Out[26]: 0.4

func(2,0)
Traceback (most recent call last):

  File "<ipython-input-27-bc999a2c372c>", line 1, in <module>
    func(2,0)

  File "<ipython-input-25-cf1f4b56882d>", line 2, in func
    assert(b!=0)

AssertionError
```

Також можна вказати повідомлення, що буде виводитись, якщо вираз є хибним.

Приклад 2

```
def func2(a,b):
    assert b!=0, 'Division by zero!'
    return a/b

func2(2,5)
Out[32]: 0.4

func2(2,0)
Traceback (most recent call last):

  File "<ipython-input-33-bc612cafdba0>", line 1, in <module>
    func2(2,0)

  File "<ipython-input-31-055d35645e1c>", line 2, in func2
    assert b!=0,'Division by zero!'

AssertionError: Division by zero!
```

8. Завдання та вправи

1. Дана послідовність цілих чисел, що закінчується числом 0. Вивести цю послідовність у зворотньому порядку. При розв'язанні цієї задачі не можна використовувати масиви та інші динамічні структури даних.
2. Які результати повернуть вказані функції:

```
def func1(a=5,b,c):
    return a+b+c

def func2(a,/,b,c=3):
    return a+b+c

def func3(a:int, b:float, *, c=5):
    return a+b+c

def func4(a, /, *, b, c):
    return a+b+c

def func5(*, a,b,c):
    return a+b+c
```

- func1(1,2,3)
- func2(1,2,3)
- func2(a=1,b=2)
- func3(a=1,b=2)
- func4(1,2,3)
- func4(1,b=2,c=3)
- func5(a=1,b=2,c=3)

3. Вказати, які змінні повинні бути параметрами, які global чи nonlocal, щоб можна було отримати вказаний результат.

```
x=1;y=2;z=3

def F1(      ):


    def f2(   ):
```

```
        x=10
        y=20
        z=30
        print('f2: x={}, y={}, z={}'.format(x, y, z))

    f2(x)
    print('F1: x={}, y={}, z={}'.format(x, y, z))

F1(x, y)
print('x={}, y={}, z={}'.format(x, y, z))
f2: x=10, y=20, z=30
F1: x=1, y=20, z=30
x=1, y=2, z=30
```

4. Встановити правильний порядок декорування для варіння картоплі та макарон.

```
def boil(f):
    def wrapper():
        print("Набрати воду у каструлю")
        print("Поставити каструлю на вогонь")
        f()
        print("Зняти з вогню після готовності")
    return wrapper

def salt(f):
    def wrapper():
        f()
        print("Посолити")
    return wrapper

def peel(f):
    def wrapper():
        print("Очистити від кожури")
        f()
    return wrapper

def potato():
    print("Кинути картоплю у каструлю")
```

```
def pasta():
   print("Кинути макарони у каструлю")

boiled_potato=#?
boiled_potato()
Набрати воду у каструлю
Поставити каструлю на вогонь
Очистити від кожури
Кинути картоплю у каструлю
Посолити
Зняти з вогню після готовності

boiled_pasta=#?
boiled_pasta()
Набрати воду у каструлю
Поставити каструлю на вогонь
Кинути макарони у каструлю
Посолити
Зняти з вогню після готовності
```

5. Змінити код так, щоб усі можливі виключення були оброблені.

```
a=int(input())
b=int(input())
print(a/b)
```

VIII Класи

1. Класи

Клас має наступну структуру:

```
class ClassName:
  def __init__(self, parameters):
      #do_something
  #other functions
```

Тут:

- **ClassName** – назва класу;
- **__init__** – конструктор;
- **self –** аналог this, вказатель на даний екземпляр класу, завдяки якому можна присвоювати змінним класу деякі значення; усі змінні екземпляру класу мають вигляд self.name, де name – деяке ім’я змінної;

- **parameters** – деякі параметри конструктору (необов'язкові).

Кожна з функцій класу повинна мати першим аргументом self. Функція **__str__**(self) викликається при друці екземпляру класу. Функція **__repr__**(self) використовується для внутрішнього подання даних.

Для створення екземпляру класу використовується ім'я класу, а у круглих дужках передаються параметри функції-конструктору (якщо вони є). Виклик функцій класу здійснюється за допомогою імені екземпляру класу.

Приклад

```
class Point:

    def __init__(self, x, y):
        self.x=x
        self.y=y

    def translate(self, dx, dy):
        self.x+=dx
        self.y+=dy

    def __str__(self):
        return ("Point at [%f, %f]" % (self.x, self.y))

    def __repr__(self):
        return ("point at (%f, %f)" % (self.x, self.y))

p1=Point(0, 0)
print(p1)
Point at [0.000000, 0.000000]

p2=Point(1, 1)

p1.translate(0.25, 1.5)

print(p1)
Point at [0.250000, 1.500000]

print(p2)
```

```
Point at [1.000000, 1.000000]
```

Якщо функція __str__ не визначена у класі, то для друку використовується подання з функції __repr__.

Приклад

```
class Point:

    def __init__(self, x, y):
        self.x=x
        self.y=y

    def translate(self, dx, dy):
        self.x+=dx
        self.y+=dy

    def __repr__(self):
        return ("point at (%f, %f)" % (self.x, self.y))

p1=Point(1,2)
print(p1)
point at (1.000000, 2.000000)
```

2. Статичні поля і функції

У класі можуть бути ***змінні*** двох типів:

- змінна **класу**, що оголошується усередині класу наступним чином:
  - classVariable=some_value,

  доступ до неї усередині функцій здійснюється так:
  - **className**.classVariable, де className – це ім'я класу;
- змінна **об'єкта** класу, що оголошується усередині конструктора або іншої функції класу наступним чином: **self.**objectVariable=some_value, доступ до неї здійснюється так само, за допомогою ключового слова self.

У класі можуть бути ***функції*** двох типів:

- звичайна функція, першим аргументом якої виступає self; доступ до неї здійсюється через ім'я об'єкта класу;
- функція класу, яка не приймає у якості аргумента self; доступ до неї здійснюється через ім'я класу;

- статичний метод, який є функцією класу та оголошується статичною одним із наступних чинов:
    - staticFunction=**staticmethod**(staticFunction);
    - **@staticmethod** – безпосередньо перед функцією;
    - **@classmethod –** у випадку використання такого декоратора декорована функція повинна приймати у якості первого аргументу самий клас.

Фунція-дестурктор має назву __del__. Видалити екземпляр класу можна наступним чином: **del** classObject, де classObject – назва екземпляру класу.

Приклад

```
class Robot:

    population=0

    def __init__(self, name):
        self.name=name
        print('New robot', self.name)
        Robot.population+=1

    def __del__(self):
        print(self.name, 'is deleted')
        Robot.population-=1
        if Robot.population==0:
            print(self.name,'was the last robot')
        else:
            print('There are {} working robots'.format(Robot.population))

    def CallRobot(self):
        print('The robot {} is working now'.format(self.name))

    def howMany():
        print('There are {} robots'.format(Robot.population))

    howMany=staticmethod(howMany)

r1=Robot('Robot1')
r1.CallRobot()
```

Robot.howMany()

r2=Robot('Robot2')
r2.CallRobot()
Robot.howMany()

del r1
del r2
Robot.howMany()

```
New robot Robot1
The robot Robot1 is working now
There are 1 robots
New robot Robot2
The robot Robot2 is working now
There are 2 robots
Robot1 is deleted
There are 1 working robots
Robot2 is deleted
Robot2 was the last robot
There are 0 robots
```

3. Властивості

а. Створення атрибутів тільки для читання

Для перетворення функції у атрибут тільки для читання, вона повинна бути продекорована декоратором @**property**.

Приклад

```
class Person(object):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return "%s %s" % (self.first_name, self.last_name)

person1=Person('Mary','Smith')
```

```
person1.first_name
Out[6]: 'Mary'

person1.last_name
Out[7]: 'Smith'

person1.full_name
Out[8]: 'Mary Smith'

person1.full_name='Jane Black'
Traceback (most recent call last):

  File "<ipython-input-9-8171ab0c2ba7>", line 1, in <module>
    person1.full_name='Jane Black'

AttributeError: can't set attribute

person1.last_name='Black'
person1.full_name
Out[11]: 'Mary Black'
```

b. Створення getter, setter

Для створення setter-функції її потрібно продекорувати декоратором @function.**setter**, де function – назва функції. При цьому обов'язково повинна бути ще одна функція з такою самою назвою, що продекорована тільки на читання декоратором @property.

Приклад

```
class Person(object):
    def __init__(self):
        self._age = None

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if isinstance(value,int) or isinstance(value,float):
```

```
            x=int(value)
            if 0<=x<=150:
                self._age =x
            else:
                raise ValueError('Incorrect age!')
        else:
            raise TypeError('Age should be integer!')

person1=Person()

person1.age="120"
Traceback (most recent call last):

  File "<ipython-input-59-70ec6385dd34>", line 1, in <module>
    person1.age="120"

  File "<ipython-input-57-a005a78fd403>", line 18, in age
    raise TypeError('Age should be integer!')

TypeError: Age should be integer!

person1.age=110.5

person1.age
Out[61]: 110

person1.age=-10
Traceback (most recent call last):

  File "<ipython-input-62-ac25820934e5>", line 1, in <module>
    person1.age=-10

  File "<ipython-input-57-a005a78fd403>", line 16, in age
    raise ValueError('Incorrect age!')

ValueError: Incorrect age!
```

4. Інкапсуляція

За замовчуванням усі члени класу є public, а усі методи virtual.

Для того, щоб змінна була **private**, її ім'я повинно починатися з подвійного знаку підкреслювання. Тоді ця змінна не буде доступною за ім'ям об'єкту класу чи класу, однак повного захисту private змінних, на жаль, немає. Вона буде доступна таким чином: objectName._className__privateName, де objectName – ім'я об'єкта класу, className – ім'я класу, privateName – ім'я private змінної. Аналогічно для private методів.

Приклад

```
class MyClass:
    __secretVarClass=5

    def __init__(self,publicVar):
        self.publicVar=publicVar
        self.__secretVar=10

    def __secretMethod(self):
        print('This is private method')

    def publicMethod(self):
        print('This is public method')

x=MyClass(22)
print('public variable=', x.publicVar)
x.publicMethod()

public variable= 22
This is public method

print('secret variable=',x.__secretVar)
Traceback (most recent call last):

  File "<ipython-input-9-898a4acfe6cc>", line 1, in <module>
    print('secret variable=',x.__secretVar)

AttributeError: 'MyClass' object has no attribute '__secretVar'
```

```
print('secret variable=',x._MyClass__secretVar)
secret variable= 10

x._MyClass__secretVar=20
print('secret variable=',x._MyClass__secretVar)
secret variable= 20

print('secret variable of class=', MyClass.__secretVarClass)
Traceback (most recent call last):

  File "<ipython-input-7-60e1c116846c>", line 1, in <module>
    print('secret variable of class=',MyClass.__secretVarClass)

AttributeError: type object 'MyClass' has no attribute '__secretVarClass'

print('secret variable of class=', MyClass._MyClass__secretVarClass)
secret variable of class= 5

MyClass._MyClass__secretVarClass=50
print('secret variable of class=', MyClass._MyClass__secretVarClass)
secret variable of class= 50

x.__secretMethod()
Traceback (most recent call last):

  File "<ipython-input-10-5ea7b8b64169>", line 1, in <module>
    x.__secretMethod()

AttributeError: 'MyClass' object has no attribute '__secretMethod'

x._MyClass__secretMethod()
This is private method
```

5. Наслідування і поліморфізм

Наслідування класу відбувається за наступною схемою:

```
class NewClass(OldClass):
    #...
```

Тут NewClass – назва нового класу, що наслідує батьківський клас OldClass.

У конструкторі класа-нащадка можна викликати конструктор батьківського класу за ім'ям базового класу чи за допомогою методу super().

У класі-нащадку можуть бути перевизначені методи батьківського класу.

Приклад

```
class SchoolMember:
   def __init__(self, name, age):
      self.name=name
      self.age=age
      print('New school member:', self.name)

   def __str__(self):
      return "Name={}, age={}".format(self.name, self.age)


class Teacher(SchoolMember):
   def __init__(self, name, age, salary):
      SchoolMember.__init__(self, name, age)
      self.salary=salary
      print('New Teacher:', self.name)

   def __str__(self):
      return SchoolMember.__str__(self)+', Salary={}'.format(self.salary)

class Student(SchoolMember):
   def __init__(self, name, age, marks):
      SchoolMember.__init__(self, name, age)
      self.marks=marks
      print('New Student:', self.name)

   def __str__(self):
      return SchoolMember.__str__(self)+', Marks={}'.format(self.marks)

t=Teacher('Ivanov I.I.', 40, 10000)
s=Student('Petrov P.P.', 20, 75)
```

```
New school member: Ivanov I.I.
New Teacher: Ivanov I.I.
New school member: Petrov P.P.
New Student: Petrov P.P.
```

```
members=[t, s]
for i in members:
   print(i)
```

```
Name=Ivanov I.I., age=40, Salary=10000
Name=Petrov P.P., age=20, Marks=75
```

а. Наслідування з інкапсуляцією

Private змінні та методи базового класу можуть бути використані у класі-нащадку через ім'я батьківського класу. Public змінні та методи базового класу можна використовувати як змінні та методи класу-нащадку через ключове слово self.

Приклад

```
class MyBasicClass:
   __secretVarClass=5

   def __init__(self,publicVar):
      self.publicVar=publicVar
      self.__secretVar=10

   def __secretMethod(self):
      print('This is private method')

   def publicMethod(self):
      print('This is public method')

class MyNewClass(MyBasicClass):

   def __init__(self,publicVar):
      MyBasicClass.__init__(self,publicVar)

   def someMethod(self):
      print('This is publicVar from MyBasicClass', self.publicVar)
```

```
        print('This is secretVarClass from MyBasicClass', MyBasicClass._MyBasicClass__secretVarClass)
        print('This is secretVar from MyBasicClass',self._MyBasicClass__secretVar)

        print('This is publicMethod from MyBasicClass:')
        self.publicMethod()
        print('This is secretMethod from MyBasicClass:')
        MyBasicClass._MyBasicClass__secretMethod(self)

x=MyNewClass(5)
x.someMethod()
This is publicVar from MyBasicClass 5
This is secretVarClass from MyBasicClass 5
This is secretVar from MyBasicClass 10
This is publicMethod from MyBasicClass:
This is public method
This is secretMethod from MyBasicClass:
This is private method

print(x.publicVar)
5

print(x._MyBasicClass__secretVar)
10
```

b. Множинне наслідування

У Python існує множинне наслідування. Клас-нащадок описується за наступною схемою:

class NewClass**(OldClass1, …, OldClassN)**:
  #...

Тут OldClass1, …, OldClassN – назви батьківських класів, NewClass – назва класа-нащадка. Доступ до конструкторів базових класів здійснюється за їхніми іменами.

Приклад

```
class A:
   def __init__(self, a):
      self.a=a
```

```
    def __str__(self):
        return "a={0}".format(self.a)

class B:
    def __init__(self, b):
        self.b=b

    def __str__(self):
        return "b={}".format(self.b)

class ABC(A, B):
    def __init__(self,a,b,c):
        A.__init__(self,a)
        B.__init__(self,b)
        self.c=c

    def __str__(self):
        return "{}, {}, c={}".format(A.__str__(self), B.__str__(self), self.c)

x=ABC(1,2,3)
print(x)
a=1, b=2, c=3
```

6. Абстрактні класи

Абстрактний клас задається за наступною схемою:

```
class MyAbstractClass(metaclass=ABCMeta):
 #...
```

Тут MyAbstractClass – назва абстрактного класу.

Перед абстрактними методами використовується декоратор **@abstractmethod**.

Для використання абстрактних класів та методів потрібно використовувати бібліотеку **abc**.

Екземпляр абстрактного класа неможливо створити, але цей клас можна використовувати для наслідування як базовий клас. При цьому якщо клас-нащадок реалізовує абстракні методи, він стає звичайним класом, а якщо ні – стає також абстрактним.

Приклад

```
from abc import *
```

```
class SchoolMember(metaclass=ABCMeta):
    def __init__(self, name, age):
        self.name=name
        self.age=age
        print('New school member:',self.name)

    @abstractmethod
    def __str__(self):
        return "Name={}, age={}".format(self.name, self.age)

m=SchoolMember("abc", 100)
Traceback (most recent call last):

  File "<ipython-input-18-cbdba7617ffb>", line 1, in <module>
    m=SchoolMember("abc",100)

TypeError: Can't instantiate abstract class SchoolMember with abstract methods __str__
```

Для абстрактних методів можна не задавати реалізацію, а, наприклад, генерувати виключення **NotImplementedError** чи використовувати оператор **pass**. Це оператор-заглушка, рівноцінний відсутності операцій.

Приклад

```
class PlaneFigure(metaclass=ABCMeta):
    @abstractmethod
    def Perimeter(self):
        raise NotImplementedError

    @abstractmethod
    def Square(self):
        pass

class Rectangle(PlaneFigure):
    def __init__(self, a, b):
        self.a=a
        self.b=b
    def Perimeter(self):
```

```
        return 2*(self.a+self.b)
    def Square(self):
        return self.a*self.b

from math import pi
class Circle(PlaneFigure):
    def __init__(self, r):
        self.r=r
    def Perimeter(self):
        return 2*pi*self.r
    def Square(self):
        return pi*self.r**2

r=Rectangle(5,10)
r.Perimeter()
Out[1]: 30

s=Circle(3)
s.Square()
Out[2]: 28.274333882308138

class Figure(PlaneFigure):
    def __init__(self, a):
        self.a=a
    def __str__(self):
        return str(a)

f=Figure(5)
Traceback (most recent call last):

  File "<ipython-input-88-378ded232c46>", line 1, in <module>
    f=Figure(5)

TypeError: Can't instantiate abstract class Figure with abstract methods
Perimeter, Square
```

7. Перевантаження операторів

Перевантаження операторів – один зі способів реалізації поліморфізму, коли можна задати свою реалізацію якого-небудь методу у своєму класі.

Список методів для перевантаження:

- **__new__**(cls[, ...]) –управляє створенням экземпляру. У якості обов'язкового арументу приймає клас (не плутати з екземпляром). Повинен повертати екземпляр класу для його подальшої передачі методу __init__.
- **__init__**(self[, ...]) – конструктор.
- **__del__**(self) – викликається при видаленні об'єкта збірником сміття.
- **__repr__**(self) – викликається вбудованою функцією repr; повертає "сирі" дані, що використовуються для внутрішнього подання в python.
- **__str__**(self) –викликається функціями str, print і format. Повертає строкове подання об'єкта.
- **__bytes__**(self) – викликається функцією bytes при перетворенні к байтам.
- **__format__**(self, format_spec) – використовується функцією format (а також методом format у строк).
- **__lt__**(self, other) – x < y викликає x.__lt__(y).
- **__le__**(self, other) – x ≤ y викликає x.__le__(y).
- **__eq__**(self, other) – x == y викликає x.__eq__(y).
- **__ne__**(self, other) – x != y викликає x.__ne__(y)
- **__gt__**(self, other) –x > y викликає x.__gt__(y).
- **__ge__**(self, other) – x ≥ y викликає x.__ge__(y).
- **__hash__**(self) – отримання хэш-суми об'єкта, наприклад, для додавання у словник.
- **__bool__**(self) – викликається при перевірці істинності. Якщо цей метод не визначений, викликається метод __len__ (об'єкти, що мають ненульову довжину, вважаються істинними).
- **__getattr__**(self, name) – викликається, коли атрибут екземпляра класа не знайдений у звичайних місцях (наприклад, у екземпляра не має методу з такою назвою).
- **__setattr__**(self, name, value) – призначення атрибута.
- **__delattr__**(self, name) – видалення атрибута (del obj.name).

- **__call__**(self[, args...]) – виклик екземпляра класа як функції.
- **__len__**(self) – довжина об’єкта.
- **__getitem__**(self, key) – доступ за індексом (або за ключем).
- **__setitem__**(self, key, value) – призначення елемента за індексом.
- **__delitem__**(self, key) – видалення елемента за індексом.
- **__iter__**(self) – повертає ітератор для контейнера.
- **__reversed__**(self) – ітератор з елементів, що йдуть у зворотньому порядку.
- **__contains__**(self, item) – перевірка на приналежність елемента контейнеру (item in self).

Перевантаження арифметичних операторів

- **__add__**(self, other) – складення x + y викликає x.__add__(y).
- **__sub__**(self, other) – віднімання (x - y).
- **__mul__**(self, other) – множення (x * y).
- **__truediv__**(self, other) – ділення (x / y).
- **__floordiv__**(self, other) – цілочисельне ділення (x // y).
- **__mod__**(self, other) – залишок від ділення (x % y).
- **__divmod__**(self, other) – частка та залишок (divmod(x, y)).
- **__pow__**(self, other[, modulo]) – підведення у ступінь (x ** y, pow(x, y[, modulo])).
- **__lshift__**(self, other) – бітовий зсув вліво (x << y).
- **__rshift__**(self, other) – бітовий зсув вправо (x >> y).
- **__and__**(self, other) – бітове I (x & y).
- **__xor__**(self, other) – бітове ВИКЛЮЧАЮЧЕ АБО (x ^ y).
- **__or__**(self, other) – бітове АБО (x | y).
- Методи
- **__radd__**(self, other),
- **__rsub__**(self, other),
- **__rmul__**(self, other),
- **__rtruediv__**(self, other),
- **__rfloordiv__**(self, other),
- **__rmod__**(self, other),
- **__rdivmod__**(self, other),
- **__rpow__**(self, other),
- **__rlshift__**(self, other),
- **__rrshift__**(self, other),

- **__rand__**(self, other),
- **__rxor__**(self, other),
- **__ror__**(self, other) – роблять те саме, що і арифметичні оператори, що перераховані вище, але для аргументів, що знаходяться справа, и тільки у тому випадку, коли для лівого операнду не визначений відповідний метод.

Наприклад, операція x + y буде спочатку намагатися викликати x.__add__(y), і тільки у тому випадку, коли це не вдасться, буде намагатися викликати y.__radd__(x). Аналогічно для інших методів.

- **__iadd__**(self, other) – +=.
- **__isub__**(self, other) – -=.
- **__imul__**(self, other) –*=.
- **__itruediv__**(self, other) – /=.
- **__ifloordiv__**(self, other) –//=.
- **__imod__**(self, other) – %=.
- **__ipow__**(self, other[, modulo]) – **=.
- **__ilshift__**(self, other) – <<=.
- **__irshift__**(self, other) – >>=.
- **__iand__**(self, other) - &=.
- **__ixor__**(self, other) – ^=.
- **__ior__**(self, other) – |=.
- **__neg__**(self) – унарний -.
- **__pos__**(self) – унарний +.
- **__abs__**(self) – модуль (abs()).
- **__invert__**(self) – інверсія (~).
- **__complex__**(self) – приведення до complex.
- **__int__**(self) – приведення до int.
- **__float__**(self) – приведення до float.
- **__round__**(self[, n]) – округлення.
- **__enter__**(self), **__exit__**(self, exc_type, exc_value, traceback) – реалізація менеджерів контексту.

Приклад

```
class Vector2D:
   def __init__(self, x, y):
      self.x=x
      self.y=y
```

```
    def __str__(self):
        return '({}, {})'.format(self.x, self.y)

    def __add__(self, other): #сложение: self + other
        return Vector2D(self.x + other.x, self.y + other.y)

    def __iadd__(self, other): #self += other
        self.x+=other.x
        self.y+=other.y
        return self

    def __sub__(self, other): #вычитание: self - other
        return Vector2D(self.x - other.x, self.y - other.y)

    def __isub__(self, other): #self -= other
        self.x-=other.x
        self.y-=other.y
        return self

    def __neg__(self): # - self
        return Vector2D(-self.x, -self.y)

x=Vector2D(2,5)
print(x)
y=Vector2D(3,8)
print('x+y=',x+y)
print('x-y',x-y)
print('-x=',-x)
x+=y
print('x=',x)
x-=y
print('x=',x)
```

```
(2, 5)
x+y= (5, 13)
x-y (-1, -3)
-x= (-2, -5)
x= (5, 13)
```

```
x= (2, 5)
```

8. Класи даних

Новий модуль **dataclasses**, що доступний, починаючи з версії Python 3.7, спрощує написання власних класів, так як спеціальні методи, такі як **__init__**(), **__repr__**() та **__eq__**() додаються автоматично.

З модуля dataclasses потрібно завантажити декоратор dataclass та функцію field, що дозволяє індентифікувати поля класу.

Декоратор **dataclass** має наступні параметри:

- **init** – чи генерувати метод __init__() автоматично, за замовчуванням init=True;
- **repr** – чи генерувати метод __repr__() автоматично за стандартною структурою, за замовчуванням repr=True;
- **eq** – чи генерувати метод порівняння __eq__(), що порівнює усі поля класу, за замовчуванням eq=True;
- **order** – чи генерувати методи порівняння __lt__(), __le__(), __gt__(), та __ge__(), що порівнюють усі поля класу в тому порядку, в якому вони записані, за замовчуванням order=False;
- **unsafe_hash** – чи генерувати метод __hash__(), за замовчуванням unsafe_hash=False;
- **frozen** – чи створювати лише екземпляри класу лише для читання, за замовчуванням frozen=False.

Функція **field** має наступні параметри:

- **default** – значення відповідного поля класу за замовчуванням;
- **default_factory** – функція без аргументів, що викликається для значення за замовчуванням, не можна одночасно задати обидва аргументи default та default_factory;
- **init** – чи включати дане поле класу у аргументи методу __init__(), за замовчуванням init=True;
- **repr** – чи використовувати дане поле класу у строковому поданні методу __repr__(), за замовчуванням repr=True;
- **hash** – чи використовувати дане поле класу у функції __hash__(), за замовчуванням hash=None;
- **compare** – чи використовувати дане поле класу у функціях порівняння, за замовчуванням compare=True;

- **metadata** – відображення за замовчуванням metadata=None.

Приклад

```
from dataclasses import dataclass, field

@dataclass(order=True)
class Region:
    name: str
    area: float
    population: int = field(repr=False, compare=False)
    covidZone: str = 'green'

    def PopulationDensity(self) ->float:
        return self.area / self.population

reg1=Region('First region', 10000, 50000)
reg1
Out[1]: Region(name='First region', area=10000, covidZone='green')

reg2=Region('Second region', 15000, 60000, 'yellow')
reg2
Out[2]: Region(name='Second region', area=15000, covidZone='yellow')

reg1<reg2
Out[3]: True

reg3=Region('First region', 5000, 100000)
reg3
Out[4]: Region(name='First region', area=5000, covidZone='green')

reg1<reg3
Out[5]: False
```

Приклад

```
@dataclass(order=True)
class Region:
    name: str
    area: float
    population: int
```

```
    covidZone: str = field(compare=False, default='green')

    def PopulationDensity(self):
        return self.area / self.population

reg4=Region('Fourth region',2000,5000)
reg4
Out[1]: Region(name='Fourth region', area=2000, population=5000,
covidZone='green')
```

При використанні класів даних можна задавати поля класу, для чого у функції field потрібно вказати **init**=False.

Приклад

```
@dataclass(order=True)
class Region:
    name: str
    area: float
    population: int = field(repr=False, compare=False)
    covidZone: str = 'green'
    country: str = field(init=False, compare=False, default='Ukraine')

    def PopulationDensity(self) ->float:
        return self.area / self.population

    @staticmethod
    def Country():
        print(Region.country)

reg=Region('Odessa', 162.42, 1015826)
reg.Country()
Ukraine

Region.Country()
Ukraine
```

9. Завдання та вправи

1. Написати клас, у якому би містилась інформація про студентів першого та другого курсів, а саме: ім’я кожного студента, його курс та оцінки, а також кількість студентів на першому та другому курсі. Окрім того, реалізувати функцію Session, яка б приймала у якості вхідного аргументу список оцінок даного студента та перевіряла би його середній бал. У випадку, якщо середній бал менше за 3, студента потрібно відрахувати, інакше – перевести на наступний курс. Якщо студент другого курсу успішно здав сесію, потрібно написати повідомлення про те, що студент отримав диплом та також відрахувати його.
2. Написати клас MyComplex, що реалізує складання, віднімання та множення комплексних чисел, а також доступ до дійсної та уявної частин.
3. Спростити написання классу MyComplex з використанням класів даних.

IX Робота з файлами

1. Робота з файлами

Для відкриття файлу використовується функція наступної сигнатури:

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
closefd=True, opener=None)

Тут:

- **file** – строка, що представляє ім’я файла;
- **mode** – визначає спосіб роботи з файлом (за замовчуванням 'rt'):
  - 'r' – читання (за замовчуванням),
  - 'w' – запис,
  - 'x' – створити новий файл та відкрити його для запису,
  - 'a' – дозапис,
  - '+' – читання та запис,
  - 'b' – бінарний режим,
  - 't' – текстовий режим (за замовчуванням).
- **buffering** – розмір буферу;
- **encoding** – кодування;
- **errors** – визначає, як оброблювати помилки:

- 'strict' – викликати ValueError виключення при помилці кодування (за замовчуванням),
- 'ignore' – ігнорувати помилки.

- **newline** – символ закінчення строки.

Для ***зчитування*** з файла можна скористатися одним з наступних способів:

- file_stream.**read**() – читання усього файла, повертає строку;
- file_stream.read(**size**) – читання перших size символів файла;
- file_stream.**readline**() – читання 1 наступної строки з файла;
- file_stream.**readlines**() – читання усього файла построчно (повертає список строк);
- конструкція **for line in file_stream** – на кожній ітерації цикла повертає 1 наступну строку файла.

Для ***запису*** у файл можна скористатися одним з наступних способів:

- file_stream.**write**(string) – запис строки string у файл;
- file_stream.**writelines**(lines) – запис списку строк у файл;
- print(string, **file**= file_stream) – запис строки string у файл.

Для закриття потоку, що пов’язаний з відкритим файлом, використовується функція file_stream.**close**().

Тут file_stream – файловий потік, що повертає функція open().

Приклад

```
f=open('texting.txt', 'w')

L=['Hello world!','Programming with Python']
for i in L:
   f.write(i + '\n')

f.close()

f=open('texting.txt', 'r')

l=[line.split() for line in f]
f.close()
print(l)
[['Hello', 'world!'], ['Programming', 'with', 'Python']]

f=open('texting.txt', 'r')
```

```
l=f.readline() #чтение 1 строки (вместе со \n)
f.close()

print(l)
Hello world!
```

Конструкцію **with** open('file.txt') **as** f**:** можна використовувати для роботи з файлом. При цьому закриття файла відбудеться навіть у випадку виникнення виключень у даному блоці.

Приклад

with open('newfile.txt', 'w') **as** f**:** #открытие файла в данной конструкции обеспечит его закрытие даже в случае возникновения исключений в данном блоке

```
x=int(input())
print('1/{}={}'.format(x, 1/x), file=f) #запись в файл
```

2. Завдання та вправи

1. За допомогою функцій роботи з файлами створити текстовий файл text.txt, куди записати строку "Hello, world!". Зберегти файл.
2. За допомогою функцій роботи з файлами відкрити текстовий файл text.txt, звідки зчитати інформацію та вивести її на консоль.

X Бібліотеки

1. Модулі

Підключення бібліотеки чи її окремих функцій можна зробити одним з наступних способів:

- **import** library – завантаження бібліотеки library;
- **import** library **as** L – завантаження бібліотеки library з використанням псевдоніму L;
- **from** library **import** function1,…, functionN – завантаження функцій function1,…, functionN з бібліотеки library;
- **from** library **import *** – завантаження усіх функцій з бібліотеки library

У випадку, якщо бібліотека була завантажена повністю, доступ до її функцій відбувається за її ім’ям або псевдонімом.

Приклад

```
import math
math.cos(0)
```

```
Out[1]: 1.0

import math as M
M.cos(0)
Out[2]: 1.0

from math import cos, sin
cos(0)
Out[3]: 1.0

sin(0)
Out[4]: 0.0

from math import *
sin(0)
Out[5]: 0.0
```

Можна написати власні бібліотеки. Їх підключення відбувається за ім'ям файла, при цьому файл бібліотеки повинен бути у тій самій папці, що і файл, який цю бібліотеку використовує.

Приклад

mymodule.py

```
def MyFunction(x):
   print("My function:",x)
```

```
import mymodule

mymodule.MyFunction(100)
My function: 100
```

Якщо потрібно використовувати програму mymodule2 не тільки як бібліотеку, але і як самостійну програму, можна виклики функцій розташувати усередині перевірки.

Приклад

mymodule2.py

```
def MyFunction(x):
   print("My function:",x)
```

```
if __name__ == '__main__':
    x=input()
    MyFunction(x)
```

import mymodule2

mymodule2.MyFunction(150) #вызывается только указанная функция

```
My function: 150
```

Приклад

import this #философия Python

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the
rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to
guess.
There should be one-- and preferably only one --
obvious way to do it.
Although that way may not be obvious at first unless
you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a
bad idea.
If the implementation is easy to explain, it may be
a good idea.
Namespaces are one honking great idea -- let's do
more of those!
```

2. Math

Математична бібліотека **math** серед інших містить наступні корисні методи:

- math.**floor**(digit) – округлення вниз;
- math.**ceil**(digit) – округлення вгору;
- math.**fabs**(digit) – модуль числа;
- math.**factorial**(digit) – факторіал;
- math.**sqrt**(digit) – квадратний корень;
- math.**pow**(digit, power) – ступінь;
- math.**exp**(digit) – експонента;
- math.**log**(digit) – натуральний логарифм;
- math.**log10**(digit) – логарифм по основі 10;
- math.**log2**(digit) – логарифм по основі 2;
- math.**sin**(digit), math.**cos**(digit), math.**tan**(digit), math.**asin**(digit), … – тригонометричні функції;
- math.**gcd**(digit1, digit2) – найбільший загальний дільник.

Зауважимо, що функції **round**(digit) **–** округлює число, та **abs**(digit) **–** модуль числа, є вбудованими Python-функціями.

Тут digit, digit1, digit2 – числа, power – ступінь.

Приклад

```
import math

x=5.3

round(x)
Out[1]: 5

math.floor(x)
Out[2]: 5

math.ceil(x)
Out[3]: 6

abs(x)
Out[4]: 5.3

math.fabs(x)
Out[5]: 5.3
```

math.**factorial**(5)
Out[6]: 120

math.**sqrt**(x)
Out[7]: 2.3021728866442674

math.**pow**(x, 4)
Out[8]: 789.0480999999999

x**4
Out[9]: 789.0480999999999

math.**exp**(x)
Out[10]: 200.33680997479166

math.**pi**
Out[11]: 3.141592653589793

math.**e**
Out[12]: 2.718281828459045

math.**log**(x)
Out[13]: 1.667706820558076

math.log(x, math.e)
Out[14]: 1.667706820558076

math.log(x, 10)
Out[15]: 0.7242758696007889

math.**log10**(x)
Out[16]: 0.724275869600789

math.log(x, 2)
Out[17]: 2.4059923596758366

math.**log2**(x)
Out[18]: 2.405992359675837

math.**sin**(x)
Out[19]: -0.8322674422239013

math.**cos**(x)
Out[20]: 0.5543743361791608

math.**tan**(x)
Out[21]: -1.50127339580693

math.**asin**(0.87)
Out[22]: 1.0552023205488061

math.**gcd**(3,5)
Out[23]: 1

math.gcd(30,5)
Out[24]: 5

3. Decimal

Бібліотека **decimal** може бути використана для більш точного подання дійсних чисел.

Конструктор **Decimal**(x) створює Decimal-число, де *x* може бути дійсним числом чи строковим поданням дійсного числа. У першому випадку число створюється разом з похибкою подання дійсних чисел, у другому випадку – точно.

getcontext().prec дозволяє встановити кількість значущих цифр у поданні дійсного числа.

Приклад

```
import decimal as D

x=D.Decimal(12.35)
x
Out[1]:
Decimal('12.34999999999999964472863211994990706443786621093375'
)

x=D.Decimal('12.35')
x
```

```
Out[2]: Decimal('12.35')

D.Decimal(1)/D.Decimal(7)
Out[3]: Decimal('0.14285714285714285714285714429')

D.getcontext().prec=5

D.Decimal(1)/D.Decimal(7)
Out[4]: Decimal('0.14286')
```

3 Decimal-чисел можна створити список, до якого можна застосувати усі основні операції над списком.

Приклад

```
l=list(map(D.Decimal, '12.5 -8.9 0.24 -43.97'.split()))
l
Out[1]: [Decimal('12.5'), Decimal('-8.9'), Decimal('0.24'), Decimal('-43.97')]

max(l)
Out[2]: Decimal('12.5')

min(l)
Out[3]: Decimal('-43.97')

sorted(l)
Out[19]: [Decimal('-43.97'), Decimal('-8.9'), Decimal('0.24'), Decimal('12.5')]

sum(l)
Out[20]: Decimal('-40.13')

x=l[0]
y=l[1]

x+y
Out[4]: Decimal('3.6')

x*y
```

Out[5]: Decimal('-111.25')

x**2
Out[6]: Decimal('156.25')

x.**sqrt**()
Out[7]: Decimal('3.5355')

x.**exp**()
Out[8]: Decimal('2.6834E+5')

x.**ln**()
Out[9]: Decimal('2.5257')

x=x.**log10**()
x
Out[10]: Decimal('1.0969')

Функція **quantize** дозволяє округлити число до заданого вигляду **exp**. Параметр **rounding** дозволяє змінити спосіб, якими проводиться округлення.

Приклад

x.**quantize**(**exp**=D.Decimal('.01')) #округление к виду exp
Out[1]: Decimal('1.10')

x.quantize(exp=D.Decimal('1'))
Out[2]: Decimal('1')

x.quantize(exp=D.Decimal('1'), **rounding**=D.ROUND_UP)
Out[3]: Decimal('2')

Функція **setcontext** дозволяє встановити розширений context.

Приклад

D.**setcontext**(D.**ExtendedContext**)

D.Decimal(1)/D.Decimal(0)
Out[1]: Decimal('Infinity')

```
D.setcontext(D.BasicContext)

D.Decimal(1)/D.Decimal(0)
Traceback (most recent call last):

  File "<ipython-input-43-f28643906932>", line 1, in <module>
    D.Decimal(1)/D.Decimal(0)

DivisionByZero: [<class 'decimal.DivisionByZero'>]

D.getcontext().prec=10
```

Приклад Віднімання близьких чисел у арифметиці з плаваючою точкою.
Нехай арифметика з плаваючою точкою визначається 4 параметрами: β=10, t=5, L=-10, U=10. Обчислимо різницю чисел 0.1234623456 і 0.1234512345.

```
x=D.Decimal('0.1234623456')
y=D.Decimal('0.1234512345')
x
Out[1]: Decimal('0.1234623456')
y
Out[2]: Decimal('0.1234512345')

Точний результат
x-y
Out[3]: Decimal('0.0000111111')
```

Приведемо до 5 значущих цифр (у даному випадку це 5 знаків після коми)

```
xf=x.quantize(exp=D.Decimal('.00001'))
xf
Out[4]: Decimal('0.12346')

yf=y.quantize(exp=D.Decimal('.00001'))
yf
Out[5]: Decimal('0.12345')
```

```
xf-yf
Out[6]: Decimal('0.00001')
```

Отримали результат з втратою 5 значущих цифр.

4. Fraction

Бібліотека **fractions** може бути використана для подання раціональних чисел.

Конструктор **Fraction**(x, y) створює дріб x/y з пари чисельник та знаменник. Якщо подати на вхід 1 параметр, то знаменник дробу буде дорівнювати 1. Також конструктор може приймати на вхід строкове подання раціонального дробу чи дійсне число, яке переводиться у раціональну дріб.

Приклад

```
import fractions as F

x=F.Fraction(5,3)
x
Out[1]: Fraction(5, 3)

F.Fraction(5)
Out[2]: Fraction(5, 1)

F.Fraction()
Out[3]: Fraction(0, 1)

F.Fraction('3/4')
Out[4]: Fraction(3, 4)

F.Fraction(0.25)
Out[5]: Fraction(1, 4)

F.Fraction('0.25')
Out[6]: Fraction(1, 4)

F.Fraction(0.333)
Out[7]: Fraction(5998794703657501, 18014398509481984)

F.Fraction('0.333')
```

Out[8]: Fraction(333, 1000)

Для раціональних чисел мають місце основні арифметичні операції.

Приклад

```
x=F.Fraction(2,5)
y=F.Fraction(1,3)

x+y
Out[1]: Fraction(11, 15)

x-y
Out[2]: Fraction(1, 15)

x*y
Out[3]: Fraction(2, 15)

x/y
Out[4]: Fraction(6, 5)

x%y
Out[5]: Fraction(1, 15)

x**y
Out[6]: 0.7368062997280773

abs(y-x)
Out[7]: Fraction(1, 15)
```

5. Random

Бібліотека **random** використовується для генерації псевдовипадкових чисел. Вона, серед інших, містить наступні функції:

- random.**random**() – повертає випадкове число з плаваючою точкою від 0 до 1;
- random.**randint**(low, high) – повертає випадкове ціле число на вказанному проміжку;
- random.**uniform**(low, high) – повертає випадкове число з плаваючою точкою на вказанному проміжку;

- random.**choice**(list) – повертає випадковий елемент послідовності (списка, строки, множини);
- random.**shuffle**(list) – перемішує елементи послідовності, змінюючи при цьому саму послідовність (працює тільки для змінних елементів).

Тут low та high – границі проміжку, list – послідовність.

Приклад

```
import random

random.random()
Out[1]: 0.2629281546667356

random.randint(1, 100)
Out[2]: 69

random.uniform(1, 100)
Out[3]: 57.90669638733465

random.choice([1, 35, 0.7, -3.8, 49.2])
Out[4]: 1

random.choice("Hello!")
Out[5]: 'e'

l=[1, 35, 0.7, -3.8, 49.2]
random.shuffle(l)
print(l)
[1, 49.2, 0.7, -3.8, 35]
```

6. Time

Бібліотека **time** використовується для роботи з часом. Основна функція **time**(), що повертає час, виражений у секундах з початку епохи.

Задяки модулю time можна оцінити, наприклад, як найбільш ефективно (за найменший час) створити простий список.

Приклад

```
import time
```

```
n=500000
st=time.time()
#1-ий варіант – створення за допомогою генератора цикла
L1=[x**2 for x in range(n)]
print("time for L1=" + str(time.time() - st))

st=time.time()
#2-ий варіант – звичайний цикл з додаванням елементів в кінець
L2=[]
for x in range(n):
    L2.append(x**2)
print("time for L2=" + str(time.time() - st))

st=time.time()
#3-ій варіант – звичайний цикл із заповненням нульового списка потрібними значеннями
L3=[0]*n
for x in range(n):
    L3[x]=x**2
print("time for L3=" + str(time.time() - st))

st=time.time()
#4-ий варіант – застосування функції до діапазону
L4=list(map(lambda x: x**2, range(n)))
print("time for L4=" + str(time.time() - st))
```

Результати запуску програми:

```
time for L1=0.1406254768371582
time for L2=0.1875
time for L3=0.1874992847442627
time for L4=0.1406257152557373

time for L1=0.1406242847442627
time for L2=0.2031252384185791
time for L3=0.1875002384185791
time for L4=0.15625

time for L1=0.1406245231628418
```

```
time for L2=0.203125
time for L3=0.1875
time for L4=0.156250238418579l

time for L1=0.140624523162841
time for L2=0.203125476837158
time for L3=0.171875
time for L4=0.156249761581420
```

Як бачимо, час варіюється в залежності від завантаження процесора, але генератор списку стабільно показує найкращий результат.

Ще більш точні результати можна отримати за допомогою функції **time.time_ns**(), яка обчислює час у наносекундах у цілому типі.

Приклад

```
def calc_time_ns(f):
    def wrapper(*args, **kwargs):
        st=time.time_ns()
        f(*args, **kwargs)
        print('Time=',time.time_ns() - st)
    return wrapper

@calc_time_ns
def CreateList1(n=500000):
    return [x**2 for x in range(n)]

@calc_time_ns
def CreateList2(n=500000):
    L2=[]
    for x in range(n):
        L2.append(x**2)
    return L2

@calc_time_ns
def CreateList3(n=500000):
    L3=[0]*n
    for x in range(n):
        L3[x]=x**2
    return L3
```

```
@calc_time_ns
def CreateList4(n=500000):
   return list(map(lambda x: x**2, range(n)))

CreateList1()
Time= 140624200

CreateList2()
Time= 156249800

CreateList3()
Time= 156250700

CreateList4()
Time= 171876400
```

7. Timeit

Щоб уникнути похибок в залежності від завантаження процесора при оцінці часу роботи функцій, можна використовувати модуль timeit.

Консольний варіант: **%timeit** expression, де expression – деякий вираз, час роботи якого потрібно оцінити. При цьому вираз expression має бути розташований в одну строку.

Приклад

```
n=500000

%timeit L1=[x**2 for x in range(n)]
152 ms ± 1.28 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

L2=[]
%timeit for x in range(n):L2.append(x**2)
180 ms ± 3.28 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

L3=[0]*n
%timeit for x in range(n):L3[x]=x**2
158 ms ± 196 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

%timeit L4=list(map(lambda x: x**2, range(n)))
```

178 ms ± 1.36 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

При використанні у редакторі потрібно завантажити бібліотеку timeit, що містить функцію timeit. Її перший параметр – строка, що представляє собой вираз, час роботи якого потрібно оцінити. Додатковий параметр – setup. Це також строка, вона представляє собою вираз, який потрібно обчислити лише 1 раз.

Приклад

```
import timeit

timeit.timeit('L1=[x**2 for x in range(100)]')
Out[1]: 22.07610032920769

timeit.timeit('for x in range(10):L2.append(x**2)', setup='L2=[]')
Out[2]: 2.8535784178153563

timeit.timeit('for x in range(100):L3[x]=x**2', setup='L3=[0]*100')
Out[3]: 24.562215081331942

timeit.timeit('L4=list(map(lambda x: x**2, range(100)))')
Out[4]: 28.288052341704486
```

Як бачимо з результатів тестування, генератор списку працює швидше за всіх, на другому місці – заповнення списку, що ініціалізований нулями.

8. Re

Регулярний вираз – це строка, що задає шаблон пошуку підстрок у тексті. Одному шаблону може відповідати багато різних строк.

Основні шаблони:

- **.** – 1 будь-який символ, окрім нової строки \n,
- **\d** – будь-яка цифра,
- **\D** – будь-який символ окрім цифри,
- **\w** – будь-яка буква або цифра, а також підкреслювання _,
- **\W** – не буква, не цифра і не підкреслювання,
- **\s** – пусте поле, будь-який пробільний символ,
- **\S** – заповнене поле (не пробільний символ),

- **\b** – границя слова (з однієї сторони буква, а з іншої – ні), відповідає позиції, а не символу,
- **\B** – не границя слова (або з обох сторін букви, або з обох сторін не букви),
- **[..]** – 1 з символів у дужках, будь-який символ діапазону: **[**+**-]** – символ + або -, [xyz] – символ x або символ y, або символ z, [a-f] – 1 з символів вказанного діапазону,...
- **[^..]** – будь-який символ, окрім перерахованих: [^a] – будь-який символ, окрім символа a,...
- ^ і **$** – початок і кінець строки відповідно;
- a|b – a або b;
- **( )** – групує вираз та повертає знайдений текст.

Квантифікатори:

- **?** - 0 або 1 входження шаблону зліва: \s? - 0 або 1 пробіл, \d? - 0 або 1 цифра, \w? - 0 або 1 буква, ...
- - 0 або більше вхождень шаблону зліва: \s* - 0 або більше пробілів, \d* - 0 або більше цифр, \w* - 0 або більше букв, ...
- **+** –- 1 або більше входжень шаблону зліва: \s+ - 1 або більше пробілів, \d+ - 1 або більше цифр, \w+ - 1 або більше букв, ...
- **{m}** – рівно m повторень: \s{3} – рівно 3 пробіла, \d{5} – рівно 5 цифр, \w{10} – рівно 10 букв, ...
- **{m,n}** – від m до n повторень включено: \s{1, 3} – від 1 до 3 пробілів,
- **{m,}** – не менше m повторень: \d{2,} – не менше 2 цифр,
- **{,n}** – не більше n повторень: \w{, 20} – не больше 20 букв, ...

За замовчуваням квантифікатори *жадібні* – захватують максимально можливу кількість символів. Додавання ? робить їх *лінивими*, вони захватують мінімально можливу кількість символів: *?, +?, ??, {m,n}?, {,n}?, {m,}?.

Бібліотека **re** використовується для роботи з регулярними виразами. Основні функції:

- re.**match**(pattern, string) – перевіряє, чи починається строка з заданого шаблону;
- match_object.**group**(0) – виводить результат – шаблон, якщо він знайдений (використовується після match);
- match_object.**start**() – повертає початкову позицію шаблону у строці;

- match_object.**end**() – повертає кінцеву позицію шаблону у строці;
- re.**search**(pattern, string) – перевіряє, чи міститься заданий шаблон у строці;
- re.**findall**(pattern, string) – повертає усі знайдені співпадіння;
- re.**split**(delimiter, string) – розділяє строку за заданим шаблоном, при цьому за допомогою параметру **maxsplit** можна встановити максимальну кількість поділів;
- re.**sub**(pattern, string) – шукає шаблон у строці і замінює його на вказану підстроку; якщо шаблон не знайдений, строка не змінюється;
- re.**compile**(string) – можна зібрати регулярний вираз у окремий об'єкт, який може бути використаний для пошуку.

Тут string – строка, pattern – шаблон, delimiter – подільник, match_object – об'єкт, що повертає функція re.match(pattern, string).

Приклад

```
import re

re.match("Hello", "Hello, world!")
Out[1]: <re.Match object; span=(0, 5), match='Hello'>

res=re.match("Hello", "Hello, world!")
res.group(0)
Out[2]: 'Hello'

print(re.match("Hi", "Hello, world!"))
None

res=re.match("Hello", "Hello, world!")
res.start()
Out[3]: 0

res.end()
Out[4]: 5

res=re.search("world", "Hello, world!")
res.group(0)
Out[5]: 'world'
```

res=re.search("world", "Hello, world! Hi, world!")
res.start()
Out[6]: 7

res.end()
Out[7]: 12

re.**findall**("world", "Hello, world! Hi, world!")
Out[8]: ['world', 'world']

re.**split**('o', 'Hello, world!')
Out[9]: ['Hell', ', w', 'rld!']

re.split(',', 'Hello, world!')
Out[10]: ['Hello', ' world!']

re.split('o', 'Hello, world!', **maxsplit**=1)
Out[11]: ['Hell', ', world!']

re.**sub**('Hello', 'Hi', 'Hello, world!')
Out[12]: 'Hi, world!'

pattern=re.**compile**("Hello")
pattern.findall("Hello, world!")
Out[13]: ['Hello']

9. Завдання та вправи

1. Обчислити у градусах кут α прямокутного трикутника по 2 катетам.
2. Повернути перше слово зі строки з використанням регулярних виразів та стандартних функцій Python.
3. Розбити строку за декількома роздільниками.

XI Допомога

1. Функції допомоги

Наступні функції використовуються у консолі, наприклад IPython:

- some_function**?** – коротка допомога;
- **help**(some_function) – детальна допомога.

str**?**

```
Init signature: str(self, /, *args, **kwargs)
Docstring:
```

```
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object.
If encoding or
errors is specified, then the object must expose a
data buffer
that will be decoded using the given encoding and
error handler.
Otherwise, returns the result of object.__str__()
(if defined)
or repr(object).
encoding defaults to sys.getdefaultencoding().
errors defaults to 'strict'.
Type:           type
```

help(str)

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |   Create a new string object from the given
object. If encoding or
 |  errors is specified, then the object must expose
a data buffer
 |  that will be decoded using the given encoding
and error handler.
 |      Otherwise,   returns   the   result   of
object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
…
```

2. Завдання та вправи

1. Прочитати допомогу для функції len.

Основи бібліотеки NumPy

NumPy – пакет для матричних обчислень. Дану бібліотеку можна імпортувати наступним чином:

import numpy as np

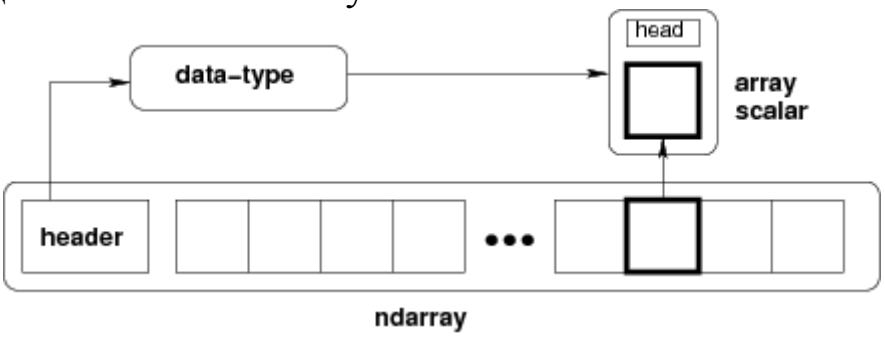
I Створення

1. Масиви ndarray

ndarray – швидкий багатовимірний масив, що споживає мало пам'яті і представляє векторні арифметичні операції.

Масиви numpy можуть містити тільки ***однотипні*** дані. ***Розмір*** масиву ***не можна змінювати*** після його створення.

Індексація елементів масиву починається з **0**.



a. Завдання за допомогою списку

Функція для створення масиву ndarray за допомогою списку має наступну сигнатуру:

np.**array**(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

Тут:

- object – список, кортеж чи подібна структура;
- dtype – тип об’єктів, що будуть зберігатися у масиві;
- copy – створити копію;
- order – порядок:
  - 'K' – збереження порядку F & C (Fortran, C++),
  - 'A' – порядок F, якщо object має порядок F, інакше – порядок C,
  - 'C' – порядок C,
  - 'F' – порядок F;
- subok – врахування підкласів;
- ndmin – мінімальна кількість вимірів масиву, що створюється.

Приклад Створення одновимірного масиву – вектор-строки, двовимірного масиву – матриці та визначення їх типів.

```
v=np.array([1, 2, 3])
v
Out[1]: array([1, 2, 3])

A=np.array([[1,2], [2,3]])
A
Out[2]:
array([[1, 2],
       [2, 3]])

type(v), type(A)
Out[3]: (numpy.ndarray, numpy.ndarray)
```

b. Завдання за допомогою функції arange

Функція для створення масиву ndarray за заданим діапазоном з певним кроком має наступну сигнатуру:

np.**arange**([start,] stop[, step,], dtype=None)

Тут:

- start – початок, за замовчуванням start= 0;
- stop – кінець (не включно);
- step – крок, за замовчуванням step=1;
- dtype – тип.

Приклад

```
x=np.arange(0, 5, 1)
x
Out[1]: array([0, 1, 2, 3, 4])

x=np.arange(0, 5)
x
Out[2]: array([0, 1, 2, 3, 4])

x=np.arange(5)
x
Out[3]: array([0, 1, 2, 3, 4])

x=np.arange(-1, 1, 0.2)
x
Out[4]:
```

```
array([ -1.00000000e+00,  -8.00000000e-01,  -6.00000000e-01,
        -4.00000000e-01,  -2.00000000e-01,  -2.22044605e-16,
         2.00000000e-01,   4.00000000e-01,   6.00000000e-01,
         8.00000000e-01])
```

c. Завдання за допомогою функції linspace

Функція для створення масиву ndarray за заданим діапазоном з певною кількістю точок має наступну сигнатуру:

np.**linspace**(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)

Тут:

- start – початок;
- stop – кінець;
- num – кількість точок, за замовчуванням – 50;
- endpoint – чи включати кінцеву точку, за замовчуванням вона включається;
- retstep – чи повертати окрім масиву крок, з яким він був побудований;
- dtype – тип;
- axis – вісь, для багатовимірних масивів, коли початок чи кінець є масивами.

Приклад

```
x=np.linspace(0, 5, 10)
x
Out[1]:
array([ 0.        ,  0.55555556,  1.11111111,  1.66666667,  2.22222222,
        2.77777778,  3.33333333,  3.88888889,  4.44444444,  5.        ])

x=np.linspace(0, 5)
x
Out[2]:
array([ 0.        ,  0.10204082,  0.20408163, ...,  4.79591837,
        4.89795918,  5.        ])
```

d. Завдання за допомогою функції logspace

Функція для створення масиву ndarray за заданим діапазоном з певною кількістю точок за логаріфмічною шкалою має наступну сигнатуру:

np.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None, axis=0)
Тут:

- start, stop, num, endpoint, dtype, axis – мають таке саме значення, що і для функції np.linspace();
- base – логаріфмічна основа, за замовчуванням base=10.

Приклад

```
np.logspace(0,10, 10, base=np.e)
Out[1]:
array([ 1.00000000e+00,  3.03773178e+00,  9.22781435e+00,
        2.80316249e+01,  8.51525577e+01,  2.58670631e+02,
        7.85771994e+02,  2.38696456e+03,  7.25095809e+03,
        2.20264658e+04])

np.exp(np.linspace(0,10,10))
Out[2]:
array([ 1.00000000e+00,  3.03773178e+00,  9.22781435e+00,
        2.80316249e+01,  8.51525577e+01,  2.58670631e+02,
        7.85771994e+02,  2.38696456e+03,  7.25095809e+03,
        2.20264658e+04])

np.logspace(0,10,10)
Out[3]:
array([ 1.00000000e+00,  1.29154967e+01,  1.66810054e+02,
        2.15443469e+03,  2.78255940e+04,  3.59381366e+05,
        4.64158883e+06,  5.99484250e+07,  7.74263683e+08,
        1.00000000e+10])

10**np.linspace(0,10,10)
Out[4]:
array([1.00000000e+00,  1.29154967e+01,  1.66810054e+02,
2.15443469e+03,
       2.78255940e+04,  3.59381366e+05,  4.64158883e+06,
5.99484250e+07,
       7.74263683e+08, 1.00000000e+10])
```

е. Завдання за допомогою mgrid

Для завдання матриць однакових форм, заповнених по горизонталі та вертикалі числами із заданих діапазонів, використовується **np.mgrid[start:stop+step:step, start:stop+step:step]**. При цьому перший діапазон визначає кількість рядків, другий – кількість стовпців.

Приклад

```
x,y=np.mgrid[0:5, 0:5]
x
Out[1]:
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]])
y
Out[2]:
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])

x, y=np.mgrid[0:5:0.5, 0:5:0.5]
x
Out[3]:
array([[ 0. ,  0. ,  0. , ...,  0. ,  0. ,  0. ],
       [ 0.5,  0.5,  0.5, ...,  0.5,  0.5,  0.5],
       [ 1. ,  1. ,  1. , ...,  1. ,  1. ,  1. ],
       ...,
       [ 3.5,  3.5,  3.5, ...,  3.5,  3.5,  3.5],
       [ 4. ,  4. ,  4. , ...,  4. ,  4. ,  4. ],
       [ 4.5,  4.5,  4.5, ...,  4.5,  4.5,  4.5]])
y
Out[4]:
array([[ 0. ,  0.5,  1. , ...,  3.5,  4. ,  4.5],
       [ 0. ,  0.5,  1. , ...,  3.5,  4. ,  4.5],
       [ 0. ,  0.5,  1. , ...,  3.5,  4. ,  4.5],
```

```
       ...,
       [ 0. ,  0.5,  1. , ...,  3.5,  4. ,  4.5],
       [ 0. ,  0.5,  1. , ...,  3.5,  4. ,  4.5],
       [ 0. ,  0.5,  1. , ...,  3.5,  4. ,  4.5]])

x, y=np.mgrid[0:2, -1:3]
x
Out[5]:
array([[0, 0, 0, 0],
       [1, 1, 1, 1]])
y
Out[6]:
array([[-1,  0,  1,  2],
       [-1,  0,  1,  2]])
```

Якщо крок задається комплексним числом, то його уявна частина визначає кількість точок або рядків/стовбців.

Приклад

```
np.mgrid[-1:1:10j]
Out[1]:
array([-1.        , -0.77777778, -0.55555556, -0.33333333, -0.11111111,
        0.11111111,  0.33333333,  0.55555556,  0.77777778,  1.        ])

np.mgrid[-1:1:4j,0:2:6j]
Out[2]:
array([[[-1.        , -1.        , -1.        , -1.        ,
         -1.        , -1.        ],
        [-0.33333333, -0.33333333, -0.33333333, -0.33333333,
         -0.33333333, -0.33333333],
        [ 0.33333333,  0.33333333,  0.33333333,  0.33333333,
          0.33333333,  0.33333333],
        [ 1.        ,  1.        ,  1.        ,  1.        ,
          1.        ,  1.        ]],

       [[ 0.        ,  0.4       ,  0.8       ,  1.2       ,
          1.6       ,  2.        ],
        [ 0.        ,  0.4       ,  0.8       ,  1.2       ,
          1.6       ,  2.        ],
```

```
[ 0.       , 0.4      , 0.8      , 1.2      ,
  1.6      , 2.       ],
[ 0.       , 0.4      , 0.8      , 1.2      ,
  1.6      , 2.       ]]])
```

f. Завдання масивів випадкових даних

Для завдання масивів випадкових даних у підмодулі **random** бібліотеки numpy існуються наступні функції:

- np.random.**rand**(rows, columns) – рівномірно розподілені числа на проміжку [0, 1);
- np.random.**randn**(rows, columns) – нормально розподілені числа з маточікуванням = 0 і дисперсією = 1;
- np.random.**randint**(low, high, size, dtype) – рівномірно розподілені числа на проміжку [`low`, `high`);
- np.random.**uniform**(low, high, size) – рівномірно розподілені числа на проміжку [`low`, `high`).

Тут rows, columns – кількість строк та стовбців, low, high – границі проміжку, size – довжина масиву або форма, dtype – тип.

Приклад

```
np.random.rand(2, 3)
Out[1]:
array([[ 0.12469257,  0.04259205,  0.6552399 ],
       [ 0.30331932,  0.95158167,  0.5782961 ]])

np.random.rand()
Out[2]: 0.8411115235779058

np.random.randn(2, 3)
Out[3]:
array([[ 0.75311277,  0.44604833, -0.17415393],
       [ 3.15075251,  1.11710818, -0.29650609]])

np.random.randn()
Out[4]: 0.9377701222581704

np.random.randint(1, 5, size=(2, 3))
Out[5]:
array([[3, 4, 1],
```

```
      [2, 3, 2]])

np.random.randint(5, size=(2, 3))
Out[6]:
array([[0, 4, 4],
       [4, 0, 0]])

np.random.randint(5)
Out[7]: 4

np.random.randint(1,5)
Out[8]: 1

np.random.uniform(1,100)
Out[9]: 32.248374409522896

np.random.uniform(1,100, size=(3,2))
Out[10]:
array([[42.38516058, 36.48265701],
       [80.15592108, 26.25260874],
       [70.89331643, 31.03872984]])
```

g. Матриці спеціальної структури

Для завдання масивів спеціальної структуру у numpy містяться наступні функції:

- np.**diag**(v, k=0) – створює з масиву/списку v діагональну матрицю з можливим зсувом відносно головної діагоналі k;
- np.**eye**(N, M=None, k=0, dtype=<class 'float'>, order='C') - одинична матриця з N строками та M стовбцями з можливим зсувом відносно головної діагоналі k типу dtype, можливо також вказати спосіб збереження у пам’яті order;
- np.**identity**(n, dtype=None) – одинична матриця з n строками та n стовбцями типу dtype;
- np.**zeros**(shape, dtype=float, order='C') – нульова матриця відповідної форми shape типу dtype, можливо також вказати спосіб збереження у пам’яті order;

- np.**ones**(shape, dtype=float, order='C') – матриця, що складається з усіх одиниць, відповідної форми shape типу dtype, можливо також вказати спосіб збереження у пам’яті order;
- np.**empty**(shape, dtype=float, order='C') – матриця з неініціалізованими елементами відповідної форми shape типу dtype, можливо також вказати спосіб збереження у пам’яті order (далеко не завжди елементи будуть нулями);
- np.**vander**(x, N=None, increasing=False) – матриця Вандермонда [x**(N-1),..., x, x**0], якщо increasing=True, то [x**0, x**1, ..., x**(N-1)].

Приклад

```
np.diag([1, 2, 3])
Out[1]:
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])

np.diag([1, 2, 3], k=1)
Out[2]:
array([[0, 1, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 3],
       [0, 0, 0, 0]])

np.diag([1, 2, 3], k=-1)
Out[3]:
array([[0, 0, 0, 0],
       [1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0]])

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

np.eye(3)
Out[5]:
```

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

np.zeros((2, 3))
Out[6]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

np.ones((2, 3))
Out[7]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])

np.empty((2, 3))
Out[8]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

np.vander(np.arange(1,5), 3)
Out[9]:
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [16,  4,  1]])
```

Створення матриці за допомогою функції np.empty є швидшим ніж, наприклад, за допомогою функції np.zeros.

Приклад

```
%timeit np.zeros((10,10))
265 ns ± 1.74 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

%timeit np.empty((10,10))
241 ns ± 1.02 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

%timeit np.zeros((100,100))
3.66 µs ± 26.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

%timeit np.empty((100,100))
456 ns ± 2.09 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

h. Створення зі строки

Для створення одновимірного масиву зі строки у numpy є функція наступної сигнатури:
np.**fromstring**(string, dtype=float, count=-1, sep='')
Тут:

- string – строка;
- dtype – тип;
- count – кількість елементів, які потрібно вибрати зі строки і помістити в масив;
- sep – роздільник між числами, що записані у строці.

Приклад

```
np.fromstring('1,-0.5,2,10', sep=',')
Out[1]: array([ 1. , -0.5,  2. , 10. ])

np.fromstring('1 2 5', dtype=int, sep=' ')
Out[2]: array([1, 2, 5])

np.fromstring('1 2 10 25', count=2, sep=' ')
Out[3]: array([1., 2.])
```

e. i. Створення з функції

Для створення масиву з функції у numpy є функція наступної сигнатури:
np.**fromfunction**(function, shape, **kwargs)
Тут:

- function – функція, що має N вхідних параметрів;
- shape – форма вихідного масиву, N – її ранг;
- dtype – тип;
- kwargs – додаткові аргументи функції function.

Приклад

```
np.fromfunction(lambda x,y: x+y, (2,3))
Out[1]:
array([[0., 1., 2.],
       [1., 2., 3.]])
```

```
np.fromfunction(lambda x,y: 10*x+y, (5,5))
Out[2]:
array([[ 0.,  1.,  2.,  3.,  4.],
       [10., 11., 12., 13., 14.],
       [20., 21., 22., 23., 24.],
       [30., 31., 32., 33., 34.],
       [40., 41., 42., 43., 44.]])
```

Створення масиву з функції є більш швидшим, ніж створення за допомогою генератору списку, але лише для масивів великих розмірностей.

Приклад

```
%timeit np.fromfunction(lambda x,y: x+y, (2,3))
13.3 µs ± 99 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit np.array([[x+y for y in range(3)] for x in range(2)])
3.67 µs ± 42.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit np.fromfunction(lambda x,y: x+y, (20,30))
15.7 µs ± 164 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit np.array([[x+y for y in range(30)] for x in range(20)])
79.6 µs ± 56.9 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

j. Порівняння по швидкості з list

Порівняймо швидкість операцій зі звичайним списком та з масивом ndarray.

Приклад

```
L=list(range(100))

%timeit [i**2 for i in L]
21.2 µs ± 76.3 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

A=np.arange(100)

%timeit A**2
725 ns ± 0.786 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Як видно, ndarray працює набагато швидше, ніж списки.

2. Matrix

Функція для створення двовимірної матриці типу matrix має наступну сигнатуру:

np.**matrix**(data, dtype=None, copy=True)

Тут:

- data – список, масив або строка;
- dtype – тип;
- copy – у випадку, коли data є масив ndarray, дані можуть копіюватися або ні.

a. Завдання за допомогою списку

Для завдання вектор-строки використовується одномірний список, а для створення матриці чи вектор-стовбця – двовимірний.

Приклад

```
v=np.matrix([1, 2, 3])
v
Out[1]: matrix([[1, 2, 3]])

M=np.matrix([[1, 2], [3, 4]])
M
Out[2]:
matrix([[1, 2],
        [3, 4]])
```

b. Завдання за допомогою строки

При створенні матриці за допомогою строки елементи в одній строці повинні бути розділені комами чи пробілами, а строки – комою з крапкою.

Приклад

```
np.matrix("1,2,3")
Out[1]: matrix([[1, 2, 3]])

np.matrix("1 2 3")
Out[2]: matrix([[1, 2, 3]])

np.matrix("1;2;3")
Out[3]:
```

```
matrix([[1],
        [2],
        [3]])

np.matrix("1,2;3,4")
Out[4]:
matrix([[1, 2],
        [3, 4]])

np.matrix("1 2;3 4")
Out[5]:
matrix([[1, 2],
        [3, 4]])
```

3. Визначення розмірів

Для визначення розмірів масивів типу ndarray чи matrix у numpy є наступні функції:

- **shape** – форма масиву у вигляді кортежу довжин відповідних розмірностей;
- **size –** кількість елементів, має додатковий параметр:
  - axis – кількість строк при axis=0, кількість стовбців при axis=1;
- **len –** кількість елементів зовнішнього виміру масиву (для векторів – кількість елементів, для двомірної матриці – кількість строк);
- **ndim –** кількість вимірів масиву;
- **itemsize –** кількість байт для кожного елемента масиву;
- **nbytes –** сумарна кількість байт для масиву.

Приклад

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

v.shape
Out[1]: (3,)

v.size
Out[2]: 3

len(v)
```

```
Out[3]: 3

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

A.shape
Out[4]: (3, 2)

A.size
Out[5]: 6

np.size(A, axis=0)
Out[6]: 3

np.size(A, axis=1)
Out[7]: 2

len(A)
Out[8]: 3

A.ndim
Out[9]: 2

A.itemsize
Out[10]: 4

A.nbytes
Out[11]: 24

M=np.matrix("1 2;3 4")

M.shape
Out[12]: (2, 2)

M.size
Out[13]: 4
```

4. Визначення типів

Для визначення типу масивів можна скористатися одною з наступних функцій:

- **type** – тип самого об’єкту (ndarray чи matrix);
- **dtype** – тип об'єктів, що містяться в масиві.

Приклад

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

type(A)
Out[1]: numpy.ndarray

A.dtype
Out[2]: dtype('int32')

M=np.matrix("1 2;3 4")

type(M)
Out[3]: numpy.matrixlib.defmatrix.matrix

M.dtype
Out[4]: dtype('int32')
```

При створенні масивів можна явно вказати тип.

Приклад

```
x=np.array([1, 2, 3], dtype=float)

x.dtype
Out[1]: dtype('float64')

x=np.array([1, 2, 3], dtype=complex)
x
Out[2]: array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

5. Перетворення типів

Для зміни типу масиву існує функція ndarray.**astype**(type), де ndarray – це масив, а type – тип, до якого потрібно перетворити ndarray. Дана функція повертає новий масив, що створюється. Дані копіюються.

Приклад

```
M=np.array([[1,4],[9,16]])
M
Out[1]:
array([[ 1,  4],
       [ 9, 16]])

M.dtype
Out[2]: dtype('int32')

MF=M.astype('float')
MF
Out[3]:
array([[  1.,   4.],
       [  9.,  16.]])

MF.dtype
Out[4]: dtype('float64')

MB=M.astype(bool)
MB
Out[5]:
array([[ True,  True],
       [ True,  True]], dtype=bool)

S=np.array(['1.25', '-9.6', '45'], dtype=np.string_)
S
Out[6]:
array([b'1.25', b'-9.6', b'45'],
      dtype='|S4')

SA=S.astype('float')
SA
Out[7]: array([  1.25,  -9.6 ,  45.  ])
```

6. Структурні масиви

Структурний масив – це об'єкт ndarray, в якому кожен елемент містить кілька іменованих полів (аналог структури).

Опис типів даних як список кортежів: назва і тип. Масив задається як список кортежів.

Звертатися до елементів структурного масиву можно за порядковими номерами строки чи за назвою стовбця через квадратні дужки. Порядок вказання номера строки та назви стовбця чи навпаки, не важливий при зверненні до елементів.

Приклад

```
arrtypes=[('x', np.float64), ('y', np.int32)]
sarr=np.array([(1, 2), (np.pi, -3)], dtype=arrtypes)
sarr
Out[1]:
array([(1.        ,  2), (3.14159265, -3)],
      dtype=[('x', '<f8'), ('y', '<i4')])

sarr[0]
Out[2]: (1., 2)

sarr[0]['x']
Out[3]: 1.0

sarr[0]['y']
Out[4]: 2

sarr['x']
Out[5]: array([1.        , 3.14159265])

sarr['x'][0]
Out[6]: 1.0

sarr[0]['x']=10

sarr['y'][1]=-30

sarr
Out[7]:
array([(10.        ,   2), ( 3.14159265, -30)],
      dtype=[('x', '<f8'), ('y', '<i4')])
```

7. Завантаження даних з файлу

Для завантаження даних з файлу можна використовувати наступні 2 функції:

- data= np.loadtxt('file.txt')
- data=np.genfromtxt('file.txt')

Основні параметри функції np.**genfromtxt**:

- **fname** – назва файлу, обов'язковий параметр;
- **dtype** – тип масиву, до якого будуть записані дані з файлу;
- **comments** – символ однострокового коментару у тексті файлу;
- **delimiter** – роздільник між даними;
- **skip_header** – кількість строк на початку файлу, що потрібно пропустити;
- **skip_footer** – кількість строк наприкінці файлу, що потрібно пропустити;
- **usecols** – послідовність цілих чисел – номерів стовбців, з яких потрібно зчитати дані;
- **max_rows** – максимальна кількість строк, що потрібно зчитати;
- **encoding** – кодування.

Приклад

'file1.csv':
a,b,c,d
1,2,3,4
5,6,7,8
9,10,11,12

```
data=np.genfromtxt('file1.csv', delimiter=',', skip_header=1)
data
Out[1]:
array([[ 1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.],
       [ 9.,  10.,  11.,  12.]])

data=np.genfromtxt('file1.csv', delimiter=',')
data
Out[2]:
array([[ nan,  nan,  nan,  nan],
       [ 1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.],
```

```
       [  9.,  10.,  11.,  12.]])

data=np.genfromtxt('file1.csv', delimiter=',', dtype=str)
data
Out[3]:
array([['a', 'b', 'c', 'd'],
       ['1', '2', '3', '4'],
       ['5', '6', '7', '8'],
       ['9', '10', '11', '12']],
      dtype='<U2')

data=np.genfromtxt('file1.csv', delimiter=',', dtype='str', max_rows=1)
Out[4]:
array(['a', 'b', 'c', 'd'],
      dtype='<U1')

data=np.genfromtxt('file1.csv', delimiter=',', skip_header=1, usecols=(0,2))
data
Out[5]:
array([[  1.,   3.],
       [  5.,   7.],
       [  9.,  11.]])
```

Для зчитування одновимірного масиву (навіть якщо в файлі буде записана матриця в декількох рядках, результатом буде її «плоский» вигляд) можна скористатися функцією np.**fromfile**, що має такі основні параметри:

- **file** – назва файлу, обов'язковий параметр;
- **dtype** – тип масиву, до якого будуть записані дані з файлу;
- **count** – кількість чисел, що потрібно зчитати;
- **sep** – роздільник між даними.

Приклад

'file2.txt':

```
1 5 -3 8 0
-6 2 18 33 10

x=np.fromfile('file2.txt', sep=' ')
x
```

```
Out[1]:
array([ 1.,  5., -3.,  8.,  0., -6.,  2., 18., 33., 10.,  9.,  0., 16.,
       -5., 24.])
```

8. Збереження даних в файл

Для збереження масиву у текстовий файл можна скористатися функцією np.**savetxt**, що має такі основні параметри:

- **file** – назва файлу, обов'язковий параметр;
- **X** – масив;
- **fmt** – строка, що описує формат даних;
- **delimiter** – роздільник між даними у стовбцях;
- **newline** – роздільник між строками;
- **header** – строка, що повинна бути записана на початку файлу;
- **footer** – строка, що повинна бути записана наприкінці файлу;
- **encoding** – кодування.

Приклад

```
A=np.random.rand(3,3)
A
Out[1]:
array([[ 0.13836992,  0.0385293 ,  0.39249992],
       [ 0.91146749,  0.65785302,  0.28039166],
       [ 0.05012698,  0.31060974,  0.70656635]])

np.savetxt("file2.csv", A)
 1.38369916690772587e-01 3.85292965367329820e-02 3.92499922609775398e-01
 9.11467487762998463e-01 6.57853016936664758e-01 2.80391663390074153e-01
 5.01269760703075828e-02 3.10609744583328684e-01 7.06566353906018296e-01

np.savetxt("file2.csv", A, fmt='%.5f')
0.13837 0.03853 0.39250
0.91147 0.65785 0.28039
0.05013 0.31061 0.70657

x=np.arange(1,10)
x
Out[2]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

np.savetxt("file2.csv", x, delimiter=',', fmt='%.2f')
1.00
2.00
```

3.00
4.00
5.00
6.00
7.00
8.00
9.00

y=np.linspace(1,5,9)
y
Out[3]: array([1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5.])

np.savetxt("file2.csv", (x,y), delimiter=';', fmt='%.2f')
1.00;2.00;3.00;4.00;5.00;6.00;7.00;8.00;9.00
1.00;1.50;2.00;2.50;3.00;3.50;4.00;4.50;5.00

9. Завдання та вправи

1. Якими трьома способами можна створити дані масиви:
   - A=array([1,2,3,4,5])
   - A=array([1,3,5,7,9])
   - A=array([0,0.5,1,1.5,2,2.5])
2. Якими функціями могли бути задані наступні масиви:
   - array([[0.22846982, 0.25925969, 0.83446707, 0.06539409],
     [0.9202888 , 0.37251142, 0.46970586, 0.43833548],
     [0.86277304, 0.90622365, 0.00493951, 0.99139339]])
   - array([[-0.66745703, 1.62988477, -0.821185 , -0.01263447],
     [-0.35998626, 0.00214046, -0.02641308, 1.12841849],
     [0.02662577, 1.52384698, -1.06466172, -1.11356025]])
   - array([[9, 7, 8, 6],
     [5, 7, 1, 6],
     [4, 9, 7, 2]])
   - array([[4.94856322, 5.69279911, 4.24952175, 1.55069265],
     [2.27747464, 1.35671247, 2.02902695, 4.18673342],
     [5.041353 , 8.49556336, 4.75011876, 4.52420315]])
3. Визначити, які створення матриць зі строки є коректними, а які – ні:
   - A=np.matrix('1,2,3;4,5,6,7')
   - A=np.matrix('1 2,3;4,5 6')
   - A=np.matrix('1 2, 3; 4, 5 6')
   - A=np.matrix("1;2;3')

4. За допомогою яких функцій можна отримати наступні розмірі матриці?

A

```
array([[-1.62866316, -1.5151682 ,  0.83730547,  2.02886996,  0.97537412,
         1.28590396],
       [ 0.47121441, -0.93893817, -1.38924895,  0.87051708,  1.41431673,
         0.02773025],
       [ 1.40975087, -0.06693909, -0.07934798,  0.38185619,  2.34659814,
         0.10585177],
       [ 0.08170984, -1.15895895, -1.56125423,  1.38466993, -1.04165768,
        -0.11156511]])
```

- (4, 6)
- 24
- 4
- 6
- 2

5. Визначити які створення вектору є коректними, а які – ні:

- x=np.array([5,-1.0,2.5], dtype=int)
- x=np.array([5,-1,2], dtype=float)
- x=np.array([5+1j,-1,2], dtype=float)
- x=np.array([5+1j,-1,2], dtype=complex)
- x=np.array(['5','-1','2'], dtype=int)

6. Для наступного файлу

fil3.txt

Name	Math	Analysis	DiffEquations	PZNO
s1	93	91	98	
s2	88	90	91	
s3	85	100	93	
s4	91	99	100	
s5	80	85	78	
s6	63	63	63	
s7	90	92	75	
s8	75	75	90	
s9	92	75	92	
s10	87	65	85	

- Зрахувати з файлу бали 10 студентів за 3 предметами
- Зрахувати з файлу бали тільки перших 5 студентів за 3 предметами

- Зрахувати з файлу бали 10 студентів тільки за ПЗНО

II Зміна, вибірка

1. Індексація

Індексація починається з 0.

Вибірка елемента, еквівалентно A[i][j]:

A[i, j], де i – номер елемента рядка, а j – номер елемента стовбця.

Вибірка і-го рядка, еквівалентно A[i].

A[i, :]

Присвоєння нового значення всьому j-му стовпцю.

A[:, j]=a, де a – будь-яке число.

Присвоєння нового значення всьому і-му рядку.

A[i, :]=b, де b – будь-яке число.

Присвоєння значень зі списку елементам другого стовпця.

A[:, 1]=[2, 3]

Присвоєння значень зі списку елементам другого рядка.

A[1, :]=[3, 4]

Приклад

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

v[1]
Out[1]: 2

v[1]=5
v
Out[2]: array([1, 5, 3])

A=np.array([[1, 2], [3, 4]])
A
Out[3]:
array([[1, 2],
       [3, 4]])

A[1, 1]
Out[4]: 4

A[1, 1]=5
A
```

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

A[:, 1]
Out[6]: array([2, 5])

A[1, :]
Out[7]: array([3, 5])

A[:, 1]=6
A
Out[8]:
array([[1, 6],
       [3, 6]])

A[1, :]=7
A
Out[9]:
array([[1, 6],
       [7, 7]])

A[:, 1]=[2, 3]
A
Out[10]:
array([[1, 2],
       [7, 3]])

A[1, :]=[3, 4]
A
Out[11]:
array([[1, 2],
       [3, 4]])
```

2. Slicing

Можна здійснювати вибірку декількох елементів за наступною схемою:

x[start:stop+step:step]

Тут:

- **start** – початковий індекс зрізу. Якщо не вказано, використовується індекс першого елемента – 0.
- **stop** – кінцевий індекс. Якщо не вказано, використовується індекс останнього елемента.
- **step** – крок вибірки. Негативне значення дозволяє будувати зріз з елементів в зворотньому порядку.

За зрізом також можна виконувати присвоювання за схемою:

- x[a:b]=list, де a – початковий індекс зрізу, b – кінцевий індекс, list – список довжини b-a.
- x[a:b]=c, де c – скаляр.

Приклад

```
x=np.arange(1,6)
x
Out[1]: array([1, 2, 3, 4, 5])

x[1:3]
Out[2]: array([2, 3])

x[1:3]=[-2,-3]
x
Out[3]: array([ 1, -2, -3,  4,  5])

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

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

x[::2]
Out[6]: array([ 1, -3,  5])

x[:3]
Out[7]: array([ 1, -2, -3])

x[3:]
Out[8]: array([4, 5])
x[-1]
```

```
Out[9]: 5

x[-3:]
Out[10]: array([-3,  4,  5])

x[1:3]=5
x
Out[11]: array([1, 5, 5, 4, 5])

A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A
Out[12]:
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])

A[1:4, 2:5]
Out[13]:
array([[12, 13, 14],
       [22, 23, 24],
       [32, 33, 34]])

A[1:3, 2]
Out[14]: array([12, 22])

A[4, 3:5]
Out[15]: array([43, 44])

A[::2, ::2]
Out[16]:
array([[ 0,  2,  4],
       [20, 22, 24],
       [40, 42, 44]])

A[::-1, ::-1]
Out[17]:
```

```
array([[44, 43, 42, 41, 40],
       [34, 33, 32, 31, 30],
       [24, 23, 22, 21, 20],
       [14, 13, 12, 11, 10],
       [ 4,  3,  2,  1,  0]])
```

Замість декількох «:» можна використовувати «...».

Приклад

```
A=np.array([[[1, 2, 3], [4, 5, 6]], [[11, 12, 13], [14, 15, 16]]])
print(A)
[[[ 1  2  3]
  [ 4  5  6]]

 [[11 12 13]
  [14 15 16]]]

A[..., 1]
Out[18]:
array([[ 2,  5],
       [12, 15]])

A[:, :, 1]
Out[19]:
array([[ 2,  5],
       [12, 15]])

A[0, ..., 2]
Out[20]: array([3, 6])

A[0, :, 2]
Out[21]: array([3, 6])
```

3. Fancy indexing

Fancy indexing – передача масиву індексів для доступу до декількох елементів масиву одночасно. В якості індексів можна використовувати списки з цілочисельними елементами. Можно використати такі виборки:

- **arr[[n, k]]** – вибирає елементи з індексами n, k де n та k – цілі числа, arr – подібний масиву об'єкт.
- **arr[[n, k, l]]** – вибирає рядки (у вказаному порядку), де arr – подібний масиву об'єкт, а n, k, l- цілі числа.
- **arr[:, [n, k, l]] –** вибірка стовпців (у вказаному порядку), де arr – подібний масиву об'єкт, а n, k, l – цілі числа.
- **arr[[n1, k1, l1], [n2, k2, l2]]** – вибірка стовпців по парам (n1,n2), (k1,k2), (l1,l2), arr – подібний масиву об'єкт.
- **arr[[n, k]] = 0** – присвоєння значення обраним рядкам (в даному випадку рядкам n, k присвоюємо значення 0), arr – подібний масиву об'єкт.
- **arr[:,[n, k]] = 1 –** присвоєння значення обраним стовпцям (в даному випадку стовпцям n, k присвоюємо значення 1), arr – подібний масиву об'єкт.

Приклад

```
x=np.arange(1,6)
x
Out[1]: array([1, 2, 3, 4, 5])

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

A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A
Out[3]:
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])

A[[1, 2, 3]]
Out[4]:
array([[10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])

A[[2, 0, 1]]
```

```
Out[5]:
array([[20, 21, 22, 23, 24],
       [ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14]])

A[:, [1, 2, 3]]
Out[6]:
array([[ 1,  2,  3],
       [11, 12, 13],
       [21, 22, 23],
       [31, 32, 33],
       [41, 42, 43]])

A[:, [3, -1, 1]]
Out[7]:
array([[ 3,  4,  1],
       [13, 14, 11],
       [23, 24, 21],
       [33, 34, 31],
       [43, 44, 41]])

A[[1, 2, 3], [1, 2, -1]]
Out[8]: array([11, 22, 34])

A[[1, 2, 3]][:,[1, 2, -1]]
Out[9]:
array([[11, 12, 14],
       [21, 22, 24],
       [31, 32, 34]])

A[[1,3,0]]=5
A
Out[10]:
array([[ 5,  5,  5,  5,  5],
       [ 5,  5,  5,  5,  5],
       [20, 21, 22, 23, 24],
       [ 5,  5,  5,  5,  5],
       [40, 41, 42, 43, 44]])
```

```
A[:,[2,-1]]=100
A
Out[11]:
array([[  5,   5, 100,   5, 100],
       [  5,   5, 100,   5, 100],
       [ 20,  21, 100,  23, 100],
       [  5,   5, 100,   5, 100],
       [ 40,  41, 100,  43, 100]])
```

4. fill

Метод numpy.ndarray.**fill()** використовується для заповнення массиву numpy скалярним значенням:

np.array.fill(**value**)

- **value**: усім елементам np.array буде присвоєно це значення

Такий самий результат можна отримати за допомогою slicing, але за довший час.

Приклад

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

x.fill(5)
print(x)
[5 5 5]

x[::]=5
print(x)
[ 5.  5.  5.]

%timeit x[::]=5
367 ns ± 1.08 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

%timeit x.fill(5)
212 ns ± 2.9 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

5. Додавання елементів

Функція numpy.**insert()** вставляє значення вздовж заданої осі перед заданими індексами. Має наступну сигнатуру:

numpy.**insert**(array, object, values, axis = None)

Параметри:

- **array:** подібний масиву об'єкт. Будь-який об'єкт, який може бути перетворений в масив NumPy.
- **object:** зріз, ціле число або послідовність цілих чисел. Визначає позицію, перед якою необхідно вставити об'єкт. У разі множинних вставок, необхідно використовувати послідовність цілих чисел.
- **values:** подібний масиву об'єкт. Значення, яке необхідно вставити в масив. При множинній вставці необхідно використовувати послідовність значень, причому довжина **values** повинна відповідати довжині послідовності **object**. Тип даних **values** завжди приводиться до типу даних масиву **array.**
- **axis:** ціле число (необов'язкове). Визначає вісь, елементи уздовж якої необхідно видалити. За замовчуванням axis = None, що відповідає стисканню вхідного масиву до однієї осі і створення так же одновимірного результуючого масиву без зазначених підмасивів.

Приклад

```
x=np.arange(5)

np.insert(x, 1, 5)
Out[1]: array([0, 5, 1, 2, 3, 4])

np.insert(x, [1,3], [10, 20])
Out[2]: array([ 0, 10,  1,  2, 20,  3,  4])

np.insert(x, [1,3], 55)
Out[3]: array([ 0, 55,  1,  2, 55,  3,  4])

np.insert(x, x[::2], 100)
Out[4]: array([100,   0,   1, 100,   2,   3, 100,   4])

np.append(x, 5)
Out[5]: array([0, 1, 2, 3, 4, 5])

np.append(x, [5,6])
Out[6]: array([0, 1, 2, 3, 4, 5, 6])

A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
```

```
A
Out[7]:
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])

np.insert(A, 1, np.arange(100,500,100))
Out[8]:
array([  0, 100, 200, 300, 400,   1,   2,   3,   4,  10,  11,  12,  13,
        14,  20,  21,  22,  23,  24,  30,  31,  32,  33,  34,  40,  41,
        42,  43,  44])

np.insert(A, 2, 1000)
Out[9]:
array([   0,    1, 1000,    2,    3,    4,   10,   11,   12,   13,   14,
         20,   21,   22,   23,   24,   30,   31,   32,   33,   34,   40,
         41,   42,   43,   44])

np.insert(A, [2,4], 1000)
Out[10]:
array([   0,    1, 1000,    2,    3, 1000,    4,   10,   11,   12,   13,
         14,   20,   21,   22,   23,   24,   30,   31,   32,   33,   34,
         40,   41,   42,   43,   44])

np.append(A, -10)
Out[11]:
array([  0,   1,   2,   3,   4,  10,  11,  12,  13,  14,  20,  21,  22,
        23,  24,  30,  31,  32,  33,  34,  40,  41,  42,  43,  44, -10])

np.append(A, [-100,-200])
Out[12]:
array([   0,    1,    2,    3,    4,   10,   11,   12,   13,   14,   20,
         21,   22,   23,   24,   30,   31,   32,   33,   34,   40,   41,
         42,   43,   44, -100, -200])
```

6. Видалення елементів

Функція numpy.**delete()** повертає новий масив із видаленням підмасивів разом із заданою віссю. Має наступну сигнатуру:

numpy.**delete**(array, object, axis = None)

Параметри:

- **array:** подібний масиву об'єкт. Будь-який об'єкт, який може бути перетворений в масив NumPy.
- **object:** зріз, ціле число або послідовність цілих чисел. Визначає позицію видаляємих підмасивів. У разі, якщо необхідно вказати зріз масиву, необхідно використовувати об'єкт зрізу, наприклад, np.s_[2: 5] або сам зріз масиву a [2: 5]. Якщо просто вказати [2: 5], то це призведе до помилки.
- **axis:** ціле число (необов'язковий). Визначає вісь, елементи уздовж якої необхідно видалити. За замовчуванням **axis = None**, що відповідає стисканню вхідного масиву до однієї осі і створення так же одновимірного результуючого масиву без зазначених підмасивів.

Приклад

```
x=np.arange(5)

np.delete(x, 2)
Out[1]: array([0, 1, 3, 4])

np.delete(x, x[::2])
Out[2]: array([1, 3])

np.delete(x, [1,3])
Out[3]: array([0, 2, 4])

A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A
Out[4]:
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

```
np.delete(A, 2)
Out[5]:
array([ 0,  1,  3,  4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31, 32,
       33, 34, 40, 41, 42, 43, 44])

np.delete(A, [1,3,10,18])
Out[6]:
array([ 0,  2,  4, 10, 11, 12, 13, 14, 21, 22, 23, 24, 30, 31, 32, 34, 40,
       41, 42, 43, 44])
```

7. Параметри друку

За замовчуванням numpy відображає тільки частину елементів масиву, якщо масив досить великий. Функція numpy.**set_printoptions()** дозволяє налаштувати параметри виведення масивів на екран. Має наступну сигнатуру:

numpy.**set_printoptions**(precision=None, threshold=None, edgeitems=None, linewidth=None, suppress=None, nanstr=None, infstr=None, formatter=None, sign=None, floatmode=None, *, legacy=None)

Параметри:

- **precision:** ціле позитивне число або **None**, (необов'язковий параметр). Задає кількість цифр після коми для чисел з плаваючою точкою. За замовчуванням **None** відповідає поточній точності (як правило 8 знаків). Використовувану за замовчуванням точність можна змінити за допомогою функції numpy.**set_printoptions**.
- **threshold:** ціле позитивне число (необов'язковий параметр). Задає загальну кількість елементів масиву, перевищення якого запускає механізм скороченого виведення. Тобто, якщо **a.size < threshold**, то масив буде виведений цілком. За замовчуванням встановлений на значення **None,** що призводить до використання поточного значення за замовчуванням (1000 елементів).
- **edgeitems:** ціле позитивне число (необов'язковий параметр). Задає кількість елементів на початку і в кінці кожного вимірювання (за замовчуванням 3).
- **linewidth:** ціле позитивне число (необов'язковий параметр). Задає кількість символів в рядку, після якого вставляється символ розриву рядків '\n' (за замовчуванням 75). У будь-якому випадку

символ перенесення буде вставлений тільки після елемента масиву.

- **suppress: True** або **False** (необов'язковий параметр). Якщо **True**, то числа з плаваючою точкою будуть виводитися як числа з фіксованою точкою, а числа, рівні нулю, в поточній точності будуть виводитися на екран як нуль. Якщо **False,** то при виведенні дійсних чисел буде використана наукова нотація у вигляді мантиси і порядку.
- **nanstr:** рядок (необов'язковий параметр). Задає строкове представлення невизначеного числа з плаваючою точкою - **NaN** (за замовчуванням '**nan**').
- **infstr:** рядок (необов'язковий параметр). Задає строкове представлення нескінченного числа з плаваючою точкою - **Inf** (за замовчуванням '**inf**').
- **sign:** один з рядків: '**+**', '**-**' або '' (порожній рядок) (необов'язковий параметр). Задає формат друку знаків для дійсних чисел: '**+**' завжди друкується знак позитивних значень; '**-**' (за замовчуванням) змушує пропускати знак позитивних значень; '' друкувати пробіл в позиції знака позитивних чисел.
- **formatter:** словник або **None** (необов'язковий параметр). Якщо **None** (за замовчуванням), то ніякого додаткового форматування до певних типів даних не застосовується. Якщо у висновку присутні певні типи даних, до яких необхідно застосувати певне форматування, то всі необхідні дії можна перерахувати в словнику. Ключами даного словника є мітки типу даних, а значеннями –необхідні функції форматування. Зазначені функції повинні повертати рядки. Типи даних, які не вказані в словнику, але присутні у виведенні, форматуються використовуваними за замовчуванням методами. Ось список окремих типів даних, до яких можна застосовувати необхідне форматування:
  - '**Bool**' – логічний (**True** і **False**);
  - '**Int**' –цілочисельний;
  - '**Timedelta**' – тип даних numpy.timedelta64;
  - '**Datetime**' – тип даних numpy.datetime64;
  - '**Float**' – речові числа (числа з плаваючою точкою);
  - '**Longfloat**' – великі речові числа 128 біт;
  - '**Complexfloat**' – комплексні числа з реальною і уявною частиною у вигляді дійсних чисел;

- **'Longcomplexfloat'** – комплексні числа з реальною і уявною частиною у вигляді дійсних чисел довжиною 128 біт;
- **'Void'** – numpy.void пусте значення;
- **'Numpystr'** – строкове представлення NumPy numpy.string_ або numpy.unicode_;
- **'Str'** – Решта рядкові формати.

Ключі визначають групу типів даних:

- **'All'** – всі типи даних;
- **'Int_kind'** /– все цілочисельні типи даних;
- **'Float_kind'** – **'float'** і **'longfloat'**;
- **'Complex_kind'** – **'complexfloat'** і **'longcomplexfloat'**;
- **'Str_kind'** – **'str'** і **'numpystr'**.

У разі, якщо функції форматування не повертають рядки, то викликається виняток TypeError.

- **floatmode:** рядок, (необов'язковий параметр). Дозволяє задати формат точності для дійсних чисел. Може приймати одне з наступних значень:
  - **'Fixed'** – друк фіксованої кількості дрібних цифр, навіть якщо ця кількість більше або менше, ніж необхідне для однозначного уявлення кожного елемента;
  - **'Unique'** – задає мінімальну кількість дрібних цифр, якої достатньо для однозначного уявлення кожного елемента;
  - **'Maxprec'** – друк з максимальною точністю, але для елементів, які можуть бути однозначно представлені з меншою кількістю дрібних цифр, друк виводиться саме з цією кількістю цифр;
  - **'Maxprec_equal'** – друк з максимальною точністю, але в разі можливості однозначного уявлення кожного елемента однаково меншою кількістю дрібних цифр використовується саме це однакове уявлення.
- **legacy:** рядок **'1.13'** або **False** (необов'язковий параметр). Якщо задано рядок **'1.13'**, то встановлюється режим друку, використовуваний NumPy до версії **1.14.0**. Якщо **False,** то застарілий режим друку відключається. Доступно в NumPy з версії **1.14.0.**

Приклад

```
import sys

np.set_printoptions(threshold= sys.maxsize)
x=np.linspace(0, 5)
x
Out[1]:
array([ 0.        ,  0.10204082,  0.20408163,  0.30612245,  0.40816327,
        0.51020408,  0.6122449 ,  0.71428571,  0.81632653,  0.91836735,
        1.02040816,  1.12244898,  1.2244898 ,  1.32653061,  1.42857143,
        1.53061224,  1.63265306,  1.73469388,  1.83673469,  1.93877551,
        2.04081633,  2.14285714,  2.24489796,  2.34693878,  2.44897959,
        2.55102041,  2.65306122,  2.75510204,  2.85714286,  2.95918367,
        3.06122449,  3.16326531,  3.26530612,  3.36734694,  3.46938776,
        3.57142857,  3.67346939,  3.7755102 ,  3.87755102,  3.97959184,
        4.08163265,  4.18367347,  4.28571429,  4.3877551 ,  4.48979592,
        4.59183673,  4.69387755,  4.79591837,  4.89795918,  5.        ])

np.set_printoptions(threshold=3)
x
Out[2]:
array([ 0.        ,  0.10204082,  0.20408163, ...,  4.79591837,
        4.89795918,  5.        ])

np.set_printoptions(precision=3)
x
Out[3]: array([ 0.   ,  0.102,  0.204, ...,  4.796,  4.898,  5.   ])

np.set_printoptions(threshold=1000, precision=8)

np.finfo(float)
Out[4]:       finfo(resolution=1e-15,       min=-1.7976931348623157e+308,
max=1.7976931348623157e+308, dtype=float64)

eps = np.finfo(float).eps
eps
Out[5]: 2.220446049250313e-16
```

```
eps = np.finfo(float).eps
x = np.arange(4.)
y = x**2 - (x + eps)**2
y
Out[6]: array([-4.93038066e-32, -4.44089210e-16,  0.00000000e+00,
0.00000000e+00])

np.set_printoptions(suppress=True)
y
Out[7]: array([-0., -0.,  0.,  0.])
```

8. Завдання та вправи

1. Як за допомогою стандартних функцій numpy можна заповнити матрицю, щоб вона прийняла даний вигляд?

```
A
array([[ 3.,  3., 10., -5.],
       [ 3.,  3.,  8., -5.],
       [ 3.,  3.,  6., -5.],
       [ 2.,  0.,  4., -5.],
       [ 0.,  2.,  2., -5.]])
```

2. Вибрати відповідні елементи з матриці.

```
A
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

- array([5, 12, 23])
- array([[5, 6, 7, 8, 9],
  [15, 16, 17, 18, 19]])
- array([[2, 4],
  [7, 9],
  [12, 14],
  [17, 19],
  [22, 24]])
- array([[10, 11, 12],
  [15, 16, 17]])

III Фільтрація

1. Вибірка по масці і за умовою

Можна вибирати ті елементи, яким відповідає значення True (False) булевого масиву. При подібних вибірках завжди створюються копії даних.

Приклад

```
x=np.arange(5)
x
Out[1]: array([0, 1, 2, 3, 4])

row_mask=np.array([True, False, False, True, False])

x[row_mask]
Out[2]: array([0, 3])
```

a. Вибірка за умовою

Можна визначити маску для вибірки, написав будь-яку умову. Тоді обираються лише ті елементи, які задовольняють умові. Якщо умов декілька, **дужки для них обов'язкові!**

Можна привласнити будь-які значення тим елементам з масиву, які задовольняють умові.

Якщо умова для 2-мірного масиву являє собою 1-мірний масив, вибираються відповідні рядки. Відповідні стовпці можна вибрати з використанням slicing.

Приклад

```
x<3
Out[1]: array([ True,  True,  True, False, False])

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

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

x[(x<3) & (x>0)] або x[(x<3) * (x>0)]
Out[4]: array([1, 2])

x[(x<3) | (x>0)] або x[(x<3) + (x>0)]
```

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

x[x<3]=33
x
Out[6]: array([33, 33, 33,  3,  4])

A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A
Out[7]:
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])

A[A<20]
Out[8]: array([ 0,  1,  2,  3,  4, 10, 11, 12, 13, 14])

A[A<10]=0
A
Out[9]:
array([[ 0,  0,  0,  0,  0],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])

A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A[A[0]>2]
Out[10]:
array([[30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])

A[:, A[0]>2]
Out[11]:
array([[ 3,  4],
       [13, 14],
       [23, 24],
```

```
      [33, 34],
      [43, 44]])

A[A[:,0]>2]
Out[12]:
array([[10, 11, 12, 13, 14],
      [20, 21, 22, 23, 24],
      [30, 31, 32, 33, 34],
      [40, 41, 42, 43, 44]])
```

b. Вибірка compress

Функція **compress**() повертає зазначені (повні) зрізи масиву вздовж заданої осі. Тобто ndarray.**compress**(condition) працює аналогічно вибірці по масці.

ndarray.**compress**(condition, **axis**=None)

- **condition** – одновимірний логічний масив який визначає критерій вибору зрізів: True – витягти; False – залишити. Якщо довжина condition менше, ніж довжина зазначеної осі масиву, то дана вісь усікається до довжини condition.
- **axis** – ціле число (необов'язковий параметр) – ось, уздовж якої витягуються зрізи. Якщо axis = None, то елементи витягуються з стисненого до однієї осі уявлення масиву a.

Для 2-мірних масивів умова має бути 1-мірною. Щоб вказати, уздовж якої осі застосовувати вибірку: 0 – вибірка рядків, 1 – вибірка стовпців.

Приклад

```
x.compress(x<3)
Out[1]: array([0, 1, 2])

x.compress((x<3) & (x>0))
Out[2]: array([1, 2])

x.compress((x<3) | (x>0))
Out[3]: array([0, 1, 2, 3, 4])

A.compress(A[0]>2, axis=1)
Out[4]:
array([[ 3,  4],
```

```
       [13, 14],
       [23, 24],
       [33, 34],
       [43, 44]])

A.compress(A[:,0]>2, axis=0)
Out[5]:
array([[10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

Метод compress працює трохи повільніше за вибірку по масці

Приклад

```
%timeit x.compress(x<3)
1.64 µs ± 14.6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

%timeit x[x<3]
1.43 µs ± 7.15 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

c. Вибірка extract

Функція **extract()** вибирає ті елементи з масиву, для яких виконується умова.

np.**extract**(condition, arr), де:

- **condition** – логічний масив, який є маскою для вихідного масиву і визначає які елементи повинні бути замінені: True (або нульове значення) – елемент витягується, False (або нульове значення) – ні.
- **arr** – вихідний масив.

Робота з 2-мірними масивами аналогічна вибірці по масці.

Приклад

```
np.extract(x<3, x)
Out[1]: array([0, 1, 2])

np.extract(A<20, A)
Out[2]: array([ 0,  1,  2,  3,  4, 10, 11, 12, 13, 14])
```

Метод extract працює повільніше за вибірку по масці і метод compress

Приклад

```
%timeit np.extract(x<3, x)
6.51 µs ± 18.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

d. Ненульові елементи nonzero

Повертає масив індексів ненульових елементів масиву x - x.**nonzero**().

Приклад

```
x.nonzero()
Out[1]: (array([1, 2, 3, 4], dtype=int32),)

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

x[x.nonzero()]
Out[3]: array([1, 2, 3, 4])

A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A[A!=0]
Out[4]:
array([ 1,  2,  3,  4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31, 32,
       33, 34, 40, 41, 42, 43, 44])

A[A.nonzero()]
Out[5]:
array([ 1,  2,  3,  4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31, 32,
       33, 34, 40, 41, 42, 43, 44])
```

Метод nonzero для вибірки ненульових елементів працює швидше, ніж вибірка по масці.

Приклад

```
%timeit x[x.nonzero()]
416 ns ± 5.47 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

%timeit x[x!=0]
1.42 µs ± 4.59 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

e. place

Функція **place()** змінює елементи масиву на зазначені значення в залежності від виконання заданих умов.

np.**place**(arr, mask, vals)

Тут:

- **arr** – вихідний масив.
- **mask** логічний масив тієї ж форми що і arr, який є маскою для вихідного масиву і визначає, які елементи повинні бути замінені: True – заміна виконується, False – елемент залишається без змін.
- **vals** – містить елементи для вставки у вихідний масив. Даний масив може бути будь-якої довжини, так як використовуються тільки перші n елементів, де n – це кількість істинних значень в mask, якщо довжина vals менше ніж n, то перебір значень для вставки буде виконуватися циклічно. Дана послідовність може бути порожньою, тільки якщо всі елементи в mask рівні False.

Приклад

```
x=np.arange(5)

np.place(x, x<3, 0)
x
Out[1]: array([0, 0, 0, 3, 4])

np.place(x, x<3, np.arange(10,50,10))
x
Out[2]: array([10, 20,  3,  3,  4])

A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
np.place(A, A<3, np.arange(100,200,10))
A
Out[3]:
array([[100, 110, 120,   3,   4],
       [ 10,  11,  12,  13,  14],
       [ 20,  21,  22,  23,  24],
       [ 30,  31,  32,  33,  34],
       [ 40,  41,  42,  43,  44]])
```

Метод place працює повільніше вибірки по масці.

Приклад

```
x=np.arange(5)
%timeit x[x<3]=100
1.05 µs ± 1.9 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

x=np.arange(5)
%timeit np.place(x, x<3, 100)
1.66 µs ± 1.77 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

2. where

a. Проста вибірка

Функція where () повертає масив індексів елементів, для яких умова вірна.

np.**where**(condition)

Можна також повертати ті елементи масиву, для яких умова вірна.

Приклад

```
x=np.array([3,-1,5,6,-2])

np.where(x>0)
Out[1]: (array([0, 2, 3], dtype=int32),)

x[np.where(x>0)]
Out[2]: array([3, 5, 6])
```

Метод where для вибірки елементів працює трохи швидше, ніж вибірка по масці

Приклад

```
%timeit x[x>0]
1.51 µs ± 3.89 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

%timeit x[np.where(x>0)]
1.48 µs ± 6.16 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

b. Вибірка з 2 масивів

Існує також така сигнатура:

np.**where**(condition, x, y)

- **condition** – логічний масив який визначає критерій вибору елементів: True – вибір елемента з x; False – вибір з y.

- **x –** масив з якого вибираються елементи якщо відповідні їм елементи в condition рівні True.
- **y** – масив з якого вибираються елементи якщо відповідні їм елементи в condition рівні False.

Можливо замість масивів у 2-му і/або 3-му аргументах використовувати скаляри.

Для випадку, коли x, y одномірні масиви, результат np.where(condition, xv, yv) еквівалентний наступному:
[xv if c else yv for (c,xv,yv) in zip(condition,x,y)]

Приклад

```
y=x*10
np.where(x>0, x, y)
Out[1]: array([  3, -10,   5,   6, -20])

A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A
Out[2]:
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])

np.where(A>10, A, -A)
Out[3]:
array([[  0,  -1,  -2,  -3,  -4],
       [-10,  11,  12,  13,  14],
       [ 20,  21,  22,  23,  24],
       [ 30,  31,  32,  33,  34],
       [ 40,  41,  42,  43,  44]])

np.where(A>10, 100, 0)
Out[4]:
array([[  0,   0,   0,   0,   0],
       [  0, 100, 100, 100, 100],
       [100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100],
```

```
       [100, 100, 100, 100, 100]])
```

Метод where як тернарна умова працює набагато швидше за генератор циклу.

Приклад

```
x=np.arange(1000)
y=np.random.randint(100, size=1000)

%timeit np.where(x>500, x, y)
4.51 µs ± 7.71 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit [i if i>500 else j for i,j in zip(x,y)]
260 µs ± 721 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

c. argwhere

Функція argwhere() повертає індекси тих елементів масиву, для яких виконується умова.

Приклад

```
x=np.array([3,-1,5,6,-2])

np.argwhere(x>0)
Out[1]:
array([[0],
       [2],
       [3]], dtype=int64)

A = np.array([[n+m*10 for n in range(5)] for m in range(5)])

np.argwhere(A>10)
Out[2]:
array([[1, 1],
       [1, 2],
       [1, 3],
       [1, 4],
       [2, 0],
       [2, 1],
       [2, 2],
       [2, 3],
```

```
[2, 4],
[3, 0],
[3, 1],
[3, 2],
[3, 3],
[3, 4],
[4, 0],
[4, 1],
[4, 2],
[4, 3],
[4, 4]], dtype=int64)
```

3. take

Функція **take ()** повертає елементи масиву з зазначеними індексами уздовж зазначеної осі.

numpy.**take**(a, indices, axis=None)

- **a** – масив NumPy або масивоподібний об'єкт. Вихідний масив.
- **indices** – масив NumPy, масивоподібний об'єкт або ціле число. Індекси видобутих елементів.
- **axis** – ціле число (необов'язковий параметр). Визначає вісь, уздовж якої витягуються елементи із зазначеним індексом. За замовчуванням axis = None, що відповідає вилученню елементів з стисненого до однієї осі уявлення масиву a.

Для масиву індексів працює аналогічно fancy indexing.

Приклад

```
v=np.arange(-3, 3)
v
Out[1]: array([-3, -2, -1,  0,  1,  2])

row_indices=[1,3,5]

v[row_indices]
Out[2]: array([-2,  0,  2])

v.take(row_indices)
Out[3]: array([-2,  0,  2])

np.take([-3, -2, -1, 0, 1, 2], row_indices)
```

```
Out[4]: array([-2,  0,  2])

A=np.random.randn(3,5)
A
Out[5]:
array([[-0.25680686,  -0.57762524,  -0.57774572,  0.57256406,
1.02932476],
       [ 0.97577829,  0.96935086,  0.90906254, -2.25284089, -0.37483638],
       [-0.36027884,  0.59674267, -0.57836124, -0.26678355,  1.55820565]])

A.take([3,1,4], axis=1)
Out[6]:
array([[ 0.57256406, -0.57762524,  1.02932476],
       [-2.25284089,  0.96935086, -0.37483638],
       [-0.26678355,  0.59674267,  1.55820565]])

A.take([2,0], axis=0)
Out[7]:
array([[-0.36027884,  0.59674267,  -0.57836124,  -0.26678355,
1.55820565],
       [-0.25680686, -0.57762524, -0.57774572,  0.57256406,  1.02932476]])
```

Метод take для вибірки елементів працює трохи швидше, ніж fancy indexing

Приклад

```
%timeit v[[1,3,5]]
3.22 µs ± 32.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit np.take(v,[1,3,5])
3.11 µs ± 13.5 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

4. put

Функція **put**() заповнює елементи з відповідними індексами (за лінеаризованою версією масиву) вказаними значеннями.

array.**put**(ind, v)

- **array** – масив NumPy або масивоподібний об'єкт. Вихідний масив.

- **v** – значення, які вставляються в зазначені індекси вихідного масиву. Значення циклічно повторюються, якщо даний список менше списку індексів. Але список може бути більше списку індексів, тоді не всі елементи будуть задіяні.

Приклад

```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A
Out[1]:
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])

A.put([1, 2, 4, 6, 10, 20], [100, 200, 300])
A
Out[2]:
array([[  0, 100, 200,   3, 300],
       [ 10, 100,  12,  13,  14],
       [200,  21,  22,  23,  24],
       [ 30,  31,  32,  33,  34],
       [300,  41,  42,  43,  44]])
```

5. choose

Функція **choose**() конструює масив вибіркою елементів з декількох масивів.

np.**choose**(which, choices)

Тут:

- **which** – масив NumPy, массівоподобний об'єкт або число. Повинен містити тільки цілі числа в інтервалі [0, n - 1], де n – це кількість масивів. Якщо вказано число, то з choices буде обраний масив з індексом, рівним цьому числу.
- **choices** – список або кортеж масивів NumPy або массівоподобних об'єктів. Набір масивів, причому всі масиви і індексний масив which повинні бути трансльований в одну і ту ж форму.

Приклад

```
which=[1,0,1,0]
choices=[[-2,-2,-2,-2], [5,5,5,5]]
```

```
np.choose(which, choices)
Out[1]: array([ 5, -2,  5, -2])

which=[1,0,1,0]
choices=[[-2,-1,0,1], [5,6,7,8]]

np.choose(which, choices)
Out[2]: array([ 5, -1,  7,  1])
```

Метод choose працює повільніше за where

Приклад

```
np.where([True,False], np.array([5,10]), np.array([1,2]))
Out[1]: array([5, 2])

np.choose([0,1], [np.array([5,10]),np.array([1,2])])
Out[2]: array([5, 2])

%timeit np.choose([0,1],[np.array([5,10]),np.array([1,2])])
12.3 µs ± 34 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit np.where([True,False],np.array([5,10]),np.array([1,2]))
5.84 µs ± 17.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

Cond=np.random.randint(0, 2, size=100)
x=np.random.rand(100)
y=np.random.rand(100)

%timeit np.choose(Cond, [x, y])
6.59 µs ± 40.9 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit np.where(1-Cond, x, y)
4.34 µs ± 22.9 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

6. select

Функція **select()** повертає масив, складений з елементів інших масивів, які вибираються з них в залежності від зазначеної умови.

np.**select**(condlist, choicelist)

- **condlist** – список логічних масивів, які виступають в ролі умов вибору елементів з масивів в choicelist. В залежності від виконання умов в condlist вибирає відповідні елементи з choicelist, якщо жодна з умов не виконується – повертає 0. Якщо значення True приймають кілька умов, то виконується тільки перше зустрінете в списку.
- **choicelist** – список тієї ж довжини, що і condlist, який містить масиви NumPy. Елементи даних масивів будуть поміщені в результуючий масив в залежності від умов в condlist.

Приклад

```
x=np.arange(10)
condlist=[x<3, x>5]
choicelist=[x, x**2]

np.select(condlist, choicelist)
Out[115]: array([ 0,  1,  2,  0,  0,  0, 36, 49, 64, 81])
```

7. diag, tril, triu

np.**diag**(A, k=0) – вилучає діагональні елементи матриці A, k – індекс діагоналі, k=0 за замовчуванням та відповідає головній діагоналі, позитивне значення k зміщує діагональ вгору, негативне – вниз.

np.**tril**(A, k=0) – витягує нижню трикутну матрицю, k – індекс діагоналі.

np.**triu**(A, k=0) – витягує верхню трикутну матрицю, k – індекс діагоналі.

Приклад

```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A
Out[1]:
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])

np.diag(A)
Out[2]: array([ 0, 11, 22, 33, 44])
```

```
np.diag(A,1)
Out[3]: array([ 1, 12, 23, 34])

np.diag(A,-1)
Out[4]: array([10, 21, 32, 43])

np.tril(A)
Out[5]:
array([[ 0,  0,  0,  0,  0],
       [10, 11,  0,  0,  0],
       [20, 21, 22,  0,  0],
       [30, 31, 32, 33,  0],
       [40, 41, 42, 43, 44]])

np.tril(A, k=1)
Out[6]:
array([[ 0,  1,  0,  0,  0],
       [10, 11, 12,  0,  0],
       [20, 21, 22, 23,  0],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])

np.tril(A, k=-1)
Out[7]:
array([[ 0,  0,  0,  0,  0],
       [10,  0,  0,  0,  0],
       [20, 21,  0,  0,  0],
       [30, 31, 32,  0,  0],
       [40, 41, 42, 43,  0]])

np.triu(A)
Out[8 ]:
array([[ 0,  1,  2,  3,  4],
       [ 0, 11, 12, 13, 14],
       [ 0,  0, 22, 23, 24],
       [ 0,  0,  0, 33, 34],
       [ 0,  0,  0,  0, 44]])
```

8. Сортування

x.**sort**() – сортує масив у порядку зростання, результат зберігається в початковому масиві.

np.**sort**(x) – сортує масив у порядку зростання, результат повертається.

np.sort(x)[::-1] – повертає масив в порядку спадання.

np.sort(A, **axis**=0) – сортує 2-мірний масив, 0 – по рядках, 1 – за стовпцями

x.**argsort**() – повертає індекси елементів у відсортованому в порядку зростання масиві.

x[x.argsort()] – елементи вихідного масиву в порядку зростання. Можна також отримати елементи іншого в порядку зростання вихідного.

np.**lexsort**((y, x)) – синхронно сортує спочатку по x, а потім по y, розміри масивів x і y повинні збігатися.

Приклад

```
x=np.array([3,-1,5,6,-2])

x.sort()
x
Out[1]: array([-2, -1,  3,  5,  6])

x=np.array([3,-1,5,6,-2])

np.sort(x)
Out[2]: array([-2, -1,  3,  5,  6])

np.sort(x)[::-1]
Out[3]: array([ 6,  5,  3, -1, -2])

A=np.array([[2,-1,5],[3,0,-8]])
A
Out[4]:
array([[ 2, -1,  5],
       [ 3,  0, -8]])

np.sort(A)
Out[5]:
```

```
array([[-1,  2,  5],
       [-8,  0,  3]])

np.sort(A, axis=0)
Out[6]:
array([[ 2, -1, -8],
       [ 3,  0,  5]])

np.sort(A, axis=1)
Out[7]:
array([[-1,  2,  5],
       [-8,  0,  3]])

ind=x.argsort()
ind
Out[8]: array([4, 1, 0, 2, 3], dtype=int32)

x[ind]
Out[9]: array([-2, -1,  3,  5,  6])

y=np.array([-1,2,7,3,0])

y[ind]
Out[10]: array([ 0,  2, -1,  7,  3])

x=np.array([5,-3,1,0,8])
y=np.array([1,-2,6,10,-4])

ind=np.lexsort((y, x))
ind
Out[11]: array([1, 3, 2, 0, 4], dtype=int32)

x[ind]
Out[12]: array([-3,  0,  1,  5,  8])

y[ind]
Out[13]: array([-2, 10,  6,  1, -4])
```

9. Випадкова перестановка

Нехай x – масив.

Розглянемо кілька функцій:

- random.**shuffle**(x) – переставляє елементи масиву x у випадковому порядку;
- random.**permutation**(x) – повертає масив з елементами масиву x, переставленими у випадковому порядку;
- random.**choice**(list) – випадковим чином обирає 1 елемент зі списку list;
- random.**choice**(list, 2) – випадковим чином обирає 2 елементи зі списку list, повертаючи масив.

Приклад

```
x = np.arange(10)
x
Out[1]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

np.random.shuffle(x)
x
Out[2]: array([5, 1, 2, 6, 8, 9, 4, 3, 7, 0])

np.random.permutation(x)
Out[3]: array([7, 9, 3, 0, 8, 5, 4, 6, 1, 2])

np.random.choice(['a','b','c','d'])
Out[4]: 'b'

np.random.choice(['a','b','c','d'], 2)
Out[5]: array(['a', 'd'], dtype='<U1')
```

10. Завдання та вправи

1. Із заданої матриці розміром 5x6 вибрати:
    - Ті строки, для яких елементи 0-го стовбця додатні, а елементи останнього стовбця від’ємні
    - Ті стовбці, для яких елементи 1-ої строки більше за 5 або елементи 3-ої строки менше 3
2. Елементи матриці, що менші -10 або більше за 20 поділити на 10, до інших елементів додати 5.

3. Від'ємні елементи матриці замінити їх абсолютним значенням, нульові елементи замінити на nan, інші елементи залишити без змін.
4. Як за допомогою стандартних функцій numpy можна заповнити матрицю, щоб вона прийняла даний вигляд?

```
A
array([[ 1.,  1.,  1., -5.,  0.,  0.],
       [ 0.,  1.,  1.,  0., -5.,  0.],
       [ 0.,  0.,  1.,  0.,  0., -5.],
       [ 2.,  6.,  0.,  3.,  0.,  0.],
       [ 0.,  2.,  6.,  3.,  3.,  0.],
       [ 0.,  0.,  2.,  3.,  3.,  3.]])
```

5. Дано 3 одновимірних масива x, y і z однакової довжини.
   - Відсортувати елементи масиву y в порядку зростання елементів масиву x
   - Відсортувати елементи масиву z в порядку спадання елементів масиву x

IV Лінійна алгебра

1. Операції зі скалярами

Оператори, до яких ми вже звикли, у цьому розділі виконують ті самі дії, що й з поодинокими числами або змінними.

Операції з векторами, матрицями та скалярами виконуються поелементно.

Нагадаємо значення операторів

- " * " – множення
- " + " – додавання
- " / " – ділення
- " - " – віднімання

Приклад

```
v=np.arange(5)
v
Out[1]: array([0, 1, 2, 3, 4])

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

2*v
Out[3]: array([0, 2, 4, 6, 8])
```

```
v+2
Out[4]: array([2, 3, 4, 5, 6])

v/2
Out[5]: array([ 0. ,  0.5,  1. ,  1.5,  2. ])

v-2
Out[6]: array([-2, -1,  0,  1,  2])

A = np.array([[n + m * 10 for n in range(5)] for m in range(5)])
A
Out[7]:
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])

A*2
Out[8]:
array([[ 0,  2,  4,  6,  8],
       [20, 22, 24, 26, 28],
       [40, 42, 44, 46, 48],
       [60, 62, 64, 66, 68],
       [80, 82, 84, 86, 88]])

A+2
Out[9]:
array([[ 2,  3,  4,  5,  6],
       [12, 13, 14, 15, 16],
       [22, 23, 24, 25, 26],
       [32, 33, 34, 35, 36],
       [42, 43, 44, 45, 46]])
```

2. Поелементні операції з np.array

Поелементні операції з np.array мають наступний вигляд

- “ + ” – поелементне додавання
- “ * ” – поелементне множення

- “ ** ” – поелементне піднесення до степеня

Приклад

```
v=np.arange(5)

v+v
Out[1]: array([0, 2, 4, 6, 8])

v*v
Out[2]: array([ 0,  1,  4,  9, 16])

v**v
Out[3]: array([  1,   1,   4,  27, 256], dtype=int32)

v**2
Out[4]: array([ 0,  1,  4,  9, 16])

A = np.array([[n+m*10 for n in range(5)] for m in range(5)])

A+A
Out[5]:
array([[ 0,  2,  4,  6,  8],
       [20, 22, 24, 26, 28],
       [40, 42, 44, 46, 48],
       [60, 62, 64, 66, 68],
       [80, 82, 84, 86, 88]])

A*A
Out[6]:
array([[   0,    1,    4,    9,   16],
       [ 100,  121,  144,  169,  196],
       [ 400,  441,  484,  529,  576],
       [ 900,  961, 1024, 1089, 1156],
       [1600, 1681, 1764, 1849, 1936]])

A**2
Out[7]:
array([[   0,    1,    4,    9,   16],
       [ 100,  121,  144,  169,  196],
```

```
 [ 400,  441,  484,  529,  576],
 [ 900,  961, 1024, 1089, 1156],
 [1600, 1681, 1764, 1849, 1936]])
```

3. Матричний добуток з np.matrix

Знайомі нам оператори можуть використовуватись не тільки поелементно, але й у матричному сенсі.

- “ * ” – матричне множення
- “ ** ” – матричне піднесення до степеня

Приклад

```
M = np.matrix([[n + m * 10 for n in range(5)] for m in range(5)])
M
Out[1]:
matrix([[ 0,  1,  2,  3,  4],
        [10, 11, 12, 13, 14],
        [20, 21, 22, 23, 24],
        [30, 31, 32, 33, 34],
        [40, 41, 42, 43, 44]])

M*M
Out[2]:
matrix([[ 300,  310,  320,  330,  340],
        [1300, 1360, 1420, 1480, 1540],
        [2300, 2410, 2520, 2630, 2740],
        [3300, 3460, 3620, 3780, 3940],
        [4300, 4510, 4720, 4930, 5140]])

M**2
Out[3]:
matrix([[ 300,  310,  320,  330,  340],
        [1300, 1360, 1420, 1480, 1540],
        [2300, 2410, 2520, 2630, 2740],
        [3300, 3460, 3620, 3780, 3940],
        [4300, 4510, 4720, 4930, 5140]])
```

У залежності від того, яким чином у матрицю записані елементи, ми можемо отримати різні виводи. І за неправильного запису елементів

при спробі перемножити отримані матриці, можемо зіткнутись із помилкою невідповідності розмірностей.

Приклад

```
vM1=np.matrix("0 1 2 3 4")
vM1
Out[1]: matrix([[0, 1, 2, 3, 4]])

vM2=np.matrix("0;1;2;3;4")
vM2
Out[2]:
matrix([[0],
        [1],
        [2],
        [3],
        [4]])

M*vM2
Out[3]:
matrix([[ 30],
        [130],
        [230],
        [330],
        [430]])

vM1*vM2
Out[4]: matrix([[30]])
```

При перемноженні M та vM1 отримаємо помилку.

Якщо для об'єктів np.matrix потрібно зробити поелементне множення, можна скористатися методом np.**multiply**.

Приклад

```
np.multiply(M, M)
Out[1]:
matrix([[   0,    1,    4,    9,   16],
        [ 100,  121,  144,  169,  196],
        [ 400,  441,  484,  529,  576],
        [ 900,  961, 1024, 1089, 1156],
        [1600, 1681, 1764, 1849, 1936]])
```

4. Матричний добуток з np.array

a. dot

За допомогою **dot()** можна виконувати матричне множення та знаходити скалярний добуток векторів.

Приклад

```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])

A.dot(A)
Out[1]:
array([[ 300,  310,  320,  330,  340],
       [1300, 1360, 1420, 1480, 1540],
       [2300, 2410, 2520, 2630, 2740],
       [3300, 3460, 3620, 3780, 3940],
       [4300, 4510, 4720, 4930, 5140]])

vA=np.arange(5)
vA
Out[2]: array([0, 1, 2, 3, 4])

A.dot(vA)
Out[3]: array([ 30, 130, 230, 330, 430])

vA.dot(vA)
Out[4]: 30
```

b. matmul

Також для перемноження матриць (векторів) можна використовувати функцію **matmul**() або **@**.

Приклад

```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])

np.matmul(A,A)
Out[1]:
array([[ 300,  310,  320,  330,  340],
       [1300, 1360, 1420, 1480, 1540],
       [2300, 2410, 2520, 2630, 2740],
       [3300, 3460, 3620, 3780, 3940],
       [4300, 4510, 4720, 4930, 5140]])
```

```
A@A
Out[2]:
array([[ 300,  310,  320,  330,  340],
       [1300, 1360, 1420, 1480, 1540],
       [2300, 2410, 2520, 2630, 2740],
       [3300, 3460, 3620, 3780, 3940],
       [4300, 4510, 4720, 4930, 5140]])

vA=np.arange(5)

np.matmul(A,vA)
Out[3]: array([ 30, 130, 230, 330, 430])

A@vA
Out[4]: array([ 30, 130, 230, 330, 430])

vA@vA
Out[5]: 30
```

Матричне множення працює швидше з функцією dot.

Приклад

```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])

%timeit A.dot(A)
1.22 µs ± 13.8 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

%timeit np.matmul(A,A)
1.84 µs ± 3.8 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

%timeit A@A
1.73 µs ± 8.1 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

5. Зовнішній і внутрішній добутки

Нехай x та y – вектори.

- за допомогою **outer**(x, y) можемо отримати добуток 2 векторів $(xy^T)_{ij}=x_iy_j$
- за допомогою **inner**(x, y) можемо отримати внутрішній (скалярний) добуток 2 векторів x^Ty

Приклад

```
x=np.array([1,2,3])
y=np.array([4,5,6])

np.outer(x, y)
Out[1]:
array([[ 4,  5,  6],
       [ 8, 10, 12],
       [12, 15, 18]])

np.inner(x, y)
Out[2]: 32

np.dot(x,y)
Out[3]: 32
```

6. Векторний добуток

Векторний добуток можна отримати, скориставшись **cross**(x,y).

Приклад

```
x=np.array([1,2,3])
y=np.array([4,5,6])

np.cross(x,y)
Out[1]: array([-3,  6, -3])
```

7. Транспонування елементів np.array

Якщо A – масив, тоді транспонувати його елементи можна наступним чином – A.**T**. Зауважимо, що для одновимірних масивів форма після транспонування не змінюється.

Приклад

```
A.T
Out[1]:
array([[ 0, 10, 20, 30, 40],
       [ 1, 11, 21, 31, 41],
       [ 2, 12, 22, 32, 42],
       [ 3, 13, 23, 33, 43],
       [ 4, 14, 24, 34, 44]])
```

```
vA.T
Out[2]: array([0, 1, 2, 3, 4])
```

8. Транспонування елементів np.matrix

Транспонування елементів np.matrix відбувається схожим чином.

Приклад

```
M.T
Out[1]:
matrix([[ 0, 10, 20, 30, 40],
        [ 1, 11, 21, 31, 41],
        [ 2, 12, 22, 32, 42],
        [ 3, 13, 23, 33, 43],
        [ 4, 14, 24, 34, 44]])

vM1.T
Out[2]:
matrix([[0],
        [1],
        [2],
        [3],
        [4]])

vM2.T
Out[3]: matrix([[0, 1, 2, 3, 4]])

vM2.T*vM2
Out[4]: matrix([[30]])

vM2.T.dot(vM2)
Out[5]: matrix([[30]])
```

9. Комплексні матриці np.array

- **conjugate**(CA) – сполучені елементи, еквівалентно CA.conjugate () і CA.**conj**()
- **real –** дійсна частина
- **imag –** уявна частина
- **abs()–** модуль комплексного числа
- **angle –** кут комплексного числа

Приклад

```
CA = np.array([[1j, 2j], [3j, 4j]])
CA
Out[1]:
array([[ 0.+1.j,  0.+2.j],
       [ 0.+3.j,  0.+4.j]])

np.conjugate(CA)
Out[2]:
array([[ 0.-1.j,  0.-2.j],
       [ 0.-3.j,  0.-4.j]])

CA.real
Out[3]:
array([[ 0.,  0.],
       [ 0.,  0.]])

CA.imag
Out[4]:
array([[ 1.,  2.],
       [ 3.,  4.]])

np.abs(CA)
Out[5]:
array([[ 1.,  2.],
       [ 3.,  4.]])

np.angle(CA+1)
Out[6]:
array([[ 0.78539816,  1.10714872],
       [ 1.24904577,  1.32581766]])
```

10. Комплексні матриці np.matrix

Зауважимо, що CM.**H** означає Hermitian conjugate: transpose + conjugate.

Приклад. Розглянемо комплексні матриці np.matrix на прикладах.

```
CM = np.matrix([[1j, 2j], [3j, 4j]])
CM
```

```
Out[1]:
matrix([[ 0.+1.j,  0.+2.j],
        [ 0.+3.j,  0.+4.j]])

np.conjugate(CM)
Out[2]:
matrix([[ 0.-1.j,  0.-2.j],
        [ 0.-3.j,  0.-4.j]])

CM.real
Out[3]:
matrix([[ 0.,  0.],
        [ 0.,  0.]])

CM.imag
Out[4]:
matrix([[ 1.,  2.],
        [ 3.,  4.]])

CM.H
CA.H - невірно!
Out[5]:
matrix([[ 0.-1.j,  0.-3.j],
        [ 0.-2.j,  0.-4.j]])

np.abs(CM)
Out[6]:
matrix([[ 1.,  2.],
        [ 3.,  4.]])

np.angle(CM+1)
Out[7]:
array([[ 0.78539816,  1.10714872],
       [ 1.24904577,  1.32581766]])
```

11. Операції лінійної алгебри np.array

Функція **linalg.inv()** обчислює обернену матрицю.

Функція **linalg.det()** обчислює визначник (детермінант) матриці.

Функція **linalg.eig()** обчислює власні числа (значення) і власні вектори квадратної матриці.

Функція **linalg.eigvals()** обчислює власні числа (значення) матриці.

Функція **linalg.norm()** обчислює норму матриці.

Функція **linalg.cond()** обчислює число обумовленості матриці. Дане число дозволяє оцінити наскільки матриця далека від матриці повного рангу або від невиродженої для квадратної матриці.

Функція **linalg.qr()** виконує **QR**-розкладання матриці. Дане розкладання дозволяє представити матрицю A у вигляді добутку **QR**, де **Q** – це ортогональна (унітарна) матриця, а **R** – це верхнетреугольна матриця.

Функція **linalg.svd()** виконує сингулярне (**SVD**) розкладання.

Функція **linalg.solve()** вирішує лінійне матричне рівняння (систему лінійних рівнянь). Ця функція обчислює значення невідомих тільки для квадратних, невироджених матриць з повним рангом, тобто тільки якщо матриця **A** розміром **{m, m}** має ранг рівний **m**. Якщо хоча б одна з цих умов не виконується, то повертається помилка **LinAlgError**.

Функція **linalg.lstsq()** вирішує завдання пошуку найменших квадратів для лінійного матричного рівняння.

Приклад

```
np.linalg.inv(CA)
Out[1]:
      array([[ 0.+2.j ,  0.-1.j ], [ 0.-1.5j,  0.+0.5j]])

np.linalg.det(CA)
Out[2]:
      (2.0000000000000004+0j)

np.linalg.eig(CA)
Out[3]:
      (array([ 0.00000000e+00-0.37228132j,  1.11022302e-
      16+5.37228132j]),
       array([[ 0.82456484 +0.00000000e+00j,  0.41597356 -5.55111512e-
      17j],
              [-0.56576746  -0.00000000e+00j,  0.90937671
       +0.00000000e+00j]]))

np.linalg.eigvals(CA)
```

```
Out[4]:
       array([          0.00000000e+00-0.37228132j,          1.11022302e-
16+5.37228132j])

np.linalg.norm(CA, ord=1)
Out[5]: 6.0

np.linalg.norm(CA, ord=2)
Out[6]: 5.4649857042190426

np.linalg.norm(CA, ord=np.inf)
Out[7]: 7.0

np.linalg.norm(CA, ord='fro')
Out[8]: 5.4772255750516612

np.linalg.cond(CA, 1)
Out[9]: 20.999999999999993

np.linalg.cond(CA, 2)
Out[14]: 14.93303437365925

np.linalg.cond(CA, np.inf)
Out[15]: 20.999999999999996

np.linalg.cond(CA, 'fro')
Out[16]: 14.999999999999998

np.linalg.qr(CA)
Out[17]:

(array([
         [  0.00000000e+00-0.31622777j,   4.99600361e-16-0.9486833j ],
         [  0.00000000e+00-0.9486833j ,  -1.11022302e-16+0.31622777j]
]),
 array([
         [-3.16227766+0.j, -4.42718872+0.j],
         [ 0.00000000+0.j, -0.63245553+0.j]
```

```
]))

np.linalg.svd(CA)
Out[18]:
        (array([
                        [      0.00000000e+00-0.40455358j,        4.99600361e-16-
0.9145143j ],
                        [       0.00000000e+00-0.9145143j    ,       -3.33066907e-
16+0.40455358j]
        ]),
         array([ 5.4649857 ,  0.36596619]),
         array([
                        [-0.57604844+0.j, -0.81741556+0.j],
                        [ 0.81741556+0.j, -0.57604844+0.j]
        ]))

A=np.array([[2.,1.,-1.], [0.,2.,1.], [1.,1.,4.]])
b=np.array([1.,1.,5.])

np.linalg.solve(A, b)
Out[19]: array([ 1.,  0.,  1.])

A=np.array([[1,2], [0,-1], [2,3]])
b=np.array([1,1,1])

np.linalg.lstsq(A, b)
Out[20]:
        (array([ 1.66666667, -0.66666667]),
         array([ 0.66666667]), 2,
         array([ 4.3218954 ,  0.56676285]))
```

12. Операции линейной алгебры np.matrix

Ще один спосіб знаходження оберненої матриці для матриць типу np.matrix: CM.**I**

Приклад

```
np.linalg.inv(CM)
Out[1]:
         matrix([
```

```
                [ 0.+2.j ,  0.-1.j ],
                [ 0.-1.5j,  0.+0.5j]
        ])

CM.I
Out[2]:
        matrix([[-0.+2.j ,  0.-1.j ], [ 0.-1.5j,  0.+0.5j]])

np.linalg.det(CM)
Out[3]: (2.0000000000000004+0j)

np.linalg.eig(CM)
Out[4]:
      (array([          0.00000000e+00-0.37228132j,          1.11022302e-
      16+5.37228132j]),
       matrix([[ 0.82456484 +0.00000000e+00j, 0.41597356 -5.55111512e-
      17j],
              [-0.56576746          -0.00000000e+00j,          0.90937671
       +0.00000000e+00j]]))

np.linalg.eigvals(CM)
Out[5]:   array([          0.00000000e+00-0.37228132j,          1.11022302e-
16+5.37228132j])

np.linalg.norm(CM)
Out[6]: 5.4772255750516612

np.linalg.cond(CM)
Out[7]: 14.933034373659252

np.linalg.qr(CM)
Out[8]:
      (matrix([[          0.00000000e+00-0.31622777j,          4.99600361e-16-
      0.9486833j ],
              [          0.00000000e+00-0.9486833j      ,          -1.11022302e-
      16+0.31622777j]]),
       matrix([[-3.16227766+0.j, -4.42718872+0.j],
              [ 0.00000000+0.j, -0.63245553+0.j]]))
```

```
np.linalg.svd(CM)
Out[9]:
        (matrix([[       0.00000000e+00-0.40455358j,          4.99600361e-16-
        0.9145143j ],
                [          0.00000000e+00-0.9145143j     ,          -3.33066907e-
        16+0.40455358j]]),
         array([ 5.4649857 ,  0.36596619]),
         matrix([[-0.57604844+0.j, -0.81741556+0.j],
                 [ 0.81741556+0.j, -0.57604844+0.j]]))

M=np.matrix([[2.,1.,-1.],[0.,2.,1.],[1.,1.,4.]])
f=np.matrix("1.;1.;5.")
np.linalg.solve(M,f)
Out[10]:
       matrix([[ 1.],
               [ 0.],
                [ 1.]])

M=np.matrix([[1,2],[0,-1],[2,3]])
f=np.matrix("1;1;1")
np.linalg.lstsq(M, f)
Out[11]:
        (matrix([[ 1.66666667],
                 [-0.66666667]]),
         matrix([[ 0.66666667]]), 2,
         array([ 4.3218954 ,  0.56676285]))
```

13. Завдання та вправи

1. Який результат отримаємо, якщо матриці A і B типу np.array і який результат, якщо типу np.matrix:

- A*B+A**B
- (A+B)*A
- 3*A*B

$$A=\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}, B=\begin{pmatrix}5 & 3\\ 2 & 1\end{pmatrix}$$

V Обчислення

1. Застосування функцій до np.array

Функція, що обчислює тригонометричний сінус елементів масива має наступну сигнатуру:

np.sin(x, *ufunc_args)

Тут:

- **x** – подібний масиву об'єкт. Масив чисел задають кут в радіанах (360 градусів дорівнюють 2π радіан);
- **ufunc_args** – аргументи універсальної функції. Аргументи, що дозволяють налаштувати і оптимізувати роботу функції.

Приклад

```
A=np.array([[1,-3,5], [2,0,4], [-2,8,6]])
A
Out[1]:
       array([[ 1, -3,  5],
              [ 2,  0,  4],
              [-2,  8,  6]])

np.sin(A)
Out[2]:
       array([[ 0.84147098, -0.14112001, -0.95892427],
              [ 0.90929743,  0.        , -0.7568025 ],
              [-0.90929743,  0.98935825, -0.2794155 ]])
```

Аналогічну сигнатуру має і функція, що обчислює експоненту всіх елементів масива: **np.exp().**

Приклад

```
np.exp(A)
Out[1]:
       array([[  2.71828183e+00,   4.97870684e-02,   1.48413159e+02],
              [  7.38905610e+00,   1.00000000e+00,   5.45981500e+01],
              [  1.35335283e-01,   2.98095799e+03,   4.03428793e+02]])

x=np.array([1.3,5.8,-9.2,11.6])
```

Функція **np.ceil()** округлює до більшого цілого числа і має таку сигнатуру:

numpy.ceil(x, *ufunc_args)

- **x** – число чи подібний масиву об'єкт;

- **ufunc_args** – аргументи універсальної функції. Аргументи, що дозволяють налаштувати і оптимізувати роботу функції.

Приклад

np.**ceil**(x)
Out[1]: array([2., 6., -9., 12.])

Функція **np.floor(x)** округлює до меншого цілого і має аналогічну сигнатуру до функції **np.ceil()**.

Приклад

np.**floor**(x)
Out[1]: array([1., 5., -10., 11.])

Функція **np.round(x)** округлює до найближчого цілого і має аналогічну сигнатуру до функції **np.ceil()**.

Приклад

np.**round**(x)
Out[1]: array([1., 6., -9., 12.])

Функція **np.modf(x)** повертає два масива: масив дробових частин і масив цілих частин. Має сигнатуру, аналогічну до функції **np.ceil()**.

Приклад

np.**modf**(x)
Out[1]: (array([0.3, 0.8, -0.2, 0.6]), array([1., 5., -9., 11.]))

x=np.array([3, np.**nan**, 5, np.**inf**, 0, -8])

Функція np**.isnan**() поелементно перевіряє чи є елемент Nan'ом, чи ні.

Функція np.**isinf**() поелементно перевіряє чи є елемент +inf, або ж -inf, чи ні.

Ці функції повертають результат у вигляді boolean масиву.

І мають сигнатуру: np.**isnan**(array [, out]) (np.**isinf**(array [, out])), де:

- array: [array_like] – вхідний масив чи об'єкт, елементи якого потрібно перевірити;
- out: [ndarray, необов'язково] – вихідний масив, типу boolean.

Приклад

np.**isnan**(x)

Out[1]: array([False, True, False, False, False, False], dtype=bool)

np.**isinf**(x)
Out[2]: array([False, False, False, True, False, False], dtype=bool)
a=np.array([1,-5,3,9])
b=np.array([-2,8,0,4])

Функція numpy.**maximum** (arr1, arr2, /, out = None) поелементно вибирає максимальні елементи, з двох масивів повертає новий масив, що містить поелементні максимуми. Якщо один з порівнюваних елементів – NaN, то цей елемент повертається. Якщо обидва елементи – NaN, то повертається перший.
А np.**minimum**(a, b) поелементно вибирає мінімальні елементи з двох масивів.
Сигнатура функцій:

- arr1: [array_like] – вхідний масив;
- arr2: [array_like] – вихідний масив;
- out: [ndarray, необов'язково] – місце, де зберігається результат.

Приклад
np.**maximum**(a, b)
Out[1]: array([1, 8, 3, 9])

np.**minimum**(a, b)
Out[2]: array([-2, -5, 0, 4])

2. min, max
x=np.array([1,-3,5,8,0,2,9])
Функція ndarray.**min**(axis=None, out=None) повертає мінімальний елемент по заданій осі. Відповідно ndarray.**max**(axis=None, out=None) повертає максимальний елемент.

- **axis** – вісь пошуку: axis = 0 – пошук по стовпчикам; axis = 1 – пошук по рядкам;
- **out**: [ndarray, необов'язково] – місце, де зберігається результат.

Приклад
x.**min**()
Out[1]: -3
x.**max**()
Out[2]: 9

Функція ndarray.**argmin**(axis=None, out=None) повертає індекс мінімального елемента. Відповідно ndarray.**argmax**(axis=None, out=None) повертає індекс максимального елемента.

- **axis** – вісь пошуку: axis = 0 – пошук по стовпчикам; axis = 1 – пошук по рядкам;
- **out** [ndarray, необов'язково] – місце, де зберігається результат.

Приклад

```
x.argmin()
Out[1]: 1

x.argmax()
Out[2]: 6

A=np.array([[1,-3,5],[2,0,4],[-2,8,6]])
A
Out[3]:
        array([[ 1, -3,  5],
               [ 2,  0,  4],
               [-2,  8,  6]])
```

Аналогічно, функції matrix.**min**(axis=None, out=None) та matrix.**min**(axis=None, out=None) знаходять відповідно мінімальний чи максимальний елемент по всій матриці.

Також matrix.**argmin**(axis=None, out=None) та matrix.**argmin**(axis=None, out=None) знаходять індекси мінімальних значень матриці по заданій осі.

Приклад

```
A.min()
Out[1]: -3

A.min(axis=0)
Out[2]: array([-2, -3,  4])

A.argmin(axis=0)
Out[3]: array([2, 0, 1], dtype=int32)

A.min(axis=1)
Out[4]: array([-3,  0, -2])
```

```
.argmin(axis=1)
Out[5]: array([1, 1, 0], dtype=int32)

A.max()
Out[6]: 8

A.max(axis=0)
Out[7]: array([2, 8, 6])

A.max(axis=1)
Out[8]: array([5, 4, 8])

A.argmax(axis=0)
Out[9]: array([1, 2, 2], dtype=int32)

A.argmax(axis=1)
Out[10]: array([2, 2, 1], dtype=int32)
```

Також завдяки використанню зрізів(slicing) можна отримати, наприклад, мінімальний/максимальний елемент по відповідним стовпчикам або строкам:

Приклад

```
A[:,1].min()
Out[1]: -3

A[2,:].max()
Out[2]: 8
```

Функція, що обмежує елементи масиву зазначеним інтервалом допустимих значень numpy.**clip**(a, a_min, a_max, out=None) повертає масив, де всі елементи, менші за значення a_min, замінюються на a_min, а всі значення, більші за a_max, замінюються на a_max.

- **a** – масив, що містить елементи для обмеження;
- **a_min** – мінімальне значення;
- **a_max** – максимальне значення;
- **out** – масив результатів.

Приклад

```
x=np.array([1,-3,5,8,0,2,9])
```

x.**clip**(**min**=0, **max**=5)
Out[1]: array([1, 0, 5, 5, 0, 2, 5])

Також функцію clip можна замінити двома присвоюваннями через вибірку за маскою.

Приклад

x[x<0]=0
x[x>5]=5
x
Out[1]: array([1, 0, 5, 5, 0, 2, 5])

Але, слід зазначити, що функція clip повільніше:

Приклад

%timeit x.clip(min=0, max=5)
17.2 µs ± 183 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit x[x<0]=0;x[x>5]=5
2.77 µs ± 15.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

Функція ndarray.**ptp**(axis = None, out = None) відіграє важливу роль у статистиці, визначає діапазон заданих чисел: ndarray.max() - ndarray.min().

- arr – вхідний масив;
- axis – вісь пошуку;
- out [ndarray, необов'язково] – інший масив, в який ми хочемо помістити результат. Масив повинен мати ті ж розміри, що і очікуваний результат.

Приклад

x.**ptp**()
Out[1]: 12

3. Статистичні операції mean, std, var

Для розрахунку статистичних характеристик числових даних у numpy містяться наступні функції:

- **np.mean**(a**,** axis=None**,** dtype=None**,** out=None**,** keepdims=<no value>, where=<no value>) – середнє значення

- **np.std**(a**,** axis=None**,** dtype=None**,** out=None**,** ddof=0**,** keepdims=<no value>, where=<no value>) – стандартне відхилення – корінь квадратний з дисперсії
- **np.var**(a, axis=None, dtype=None, out=None, ddof=0, keepdims=<no value>, where=<no value>) – дисперсія

Тут:

- a – масив чисел, середнє значення яких потрібно розрахувати, axis – вісь, по якій обчислюється значення,
- dtype – тип даних,
- out – альтернативний масив, в якому розміщується результат,
- ddof – кількість елементів,
- keepdims – якщо вставлено значення True, вісі залишаться розміром один,
- where – для яких елементів обчислювати.

Приклад

```
x=np.array([1,-3,5,8,0,2,9])

x.mean()
Out[1]: 3.1428571428571428

x.mean(where=(x>0))
Out[2]: 5.0

x.std()
Out[3]: 4.0506991082165218

x.std(where=(x>0))
Out[4]: 3.1622776601683795

x.var()
Out[5]: 16.408163265306122

(x**2).mean()-(x.mean())**2
Out[6]: 16.408163265306122

x.var(where=(x>0))
Out[7]: 10.0
```

```
np.median(x)
Out[8]: 2.0

A=np.array([[1,-3,5],[2,0,4],[-2,8,6]])
A
Out[9]:
array([[ 1, -3,  5],
       [ 2,  0,  4],
       [-2,  8,  6]])

A.mean()
Out[10]: 2.3333333333333335

A.mean(where=(A<5))
Out[11]: 0.3333333333333333

A.mean(axis=0)
Out[12]: array([ 0.33333333,  1.66666667,  5.        ])

A.mean(axis=0, where=(A<5))
Out[13]: array([ 0.33333333, -1.5       ,  4.        ])

A.mean(axis=1)
Out[14]: array([ 1.,  2.,  4.])

A.std()
Out[15]: 3.496029493900505

A.std(axis=0)
Out[16]: array([ 1.69967317,  4.64279609,  0.81649658])

A.std(axis=1)
Out[17]: array([ 3.26598632,  1.63299316,  4.3204938 ])

A.var()
Out[18]: 12.222222222222221

A.var(axis=0)
```

Out[19]: array([2.88888889, 21.55555556, 0.66666667])

A.var(axis=1)
Out[20]: array([10.66666667, 2.66666667, 18.66666667])

4. sum, prod, trace

Для таких математичних операцій з масивами як додавання та множення у numpy містяться наступні функції:

- **np.sum**(a**,** axis=None**,** dtype=None**,** out=None**,** keepdims=<no value>**,** initial=<no value>**,** where=<no value>) – сума елементів масива.
- **np.cumsum**(a, axis=None, dtype = None, out=None) – повертає сукупну суму елементів.
- **np.prod**(a**,** axis=None**,** dtype=None**,** out=None**,** keepdims=<no value>**,** initial=<no value>**,** where=<no value>) – повертає добуток элементів масива.
- **np.cumprod**(a , axis = None , dtype = None , out = None) – сукупний добуток елементів.
- **np.trace**(a**,** offset=0**,** axis1=0**,** axis2=1**,** dtype=None**,** out=None) – сума по діагоналях масива

Тут:

- a – елементи матриці,
- offset – зсув діагоналі від головної діагоналі,
- axis – осі, за якими проводиться сума,
- dtype – тип масиву, що повертається,
- out – альтернативний масив, в якому розміщується результат,
- keepdims – якщо вставлено значення True, вісі залишаться розміром один,
- initial – початкове значення суми,
- where – елементи, що включається до суми.

Приклад

x=np.array([1,-3,5,8,0,2,9])

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

```
            [-2,  8,  6]])

x.sum()
Out[2]: 22

x.sum(where=(x>0))
Out[3]: 25

A.sum()
Out[4]: 21

A.sum(where=(A>0))
Out[5]: 26

A.sum(axis=0)
Out[6]: array([ 1,  5, 15])

A.sum(axis=1)
Out[7]: array([ 3,  6, 12])

x.cumsum()
Out[8]: array([ 1, -2,  3, 11, 11, 13, 22], dtype=int32)

A.cumsum()
Out[9]: array([ 1, -2,  3,  5,  5,  9,  7, 15, 21], dtype=int32)

A.cumsum(axis=0)
Out[10]:
        array([[ 1, -3,  5],
               [ 3, -3,  9],
               [ 1,  5, 15]], dtype=int32)

A.cumsum(axis=1)
Out[11]:
        array([[ 1, -2,  3],
               [ 2,  2,  6],
               [-2,  6, 12]], dtype=int32)
```

```
x.prod()
Out[12]: 0

x.prod(where=(x>0))
Out[13]: 720

A.prod()
Out[14]: 0

A.prod(where=(A>0))
Out[15]: 1920

A.prod(axis=0)
Out[16]: array([ -4,   0, 120])

A.prod(axis=1)
Out[17]: array([-15,   0, -96])

x.cumprod()
Out[18]: array([   1,   -3,  -15, -120,    0,    0,    0], dtype=int32)

A.cumprod()
Out[19]: array([  1,  -3, -15, -30,   0,   0,   0,   0,   0], dtype=int32)

A.cumprod(axis=0)
Out[20]:
        array([[  1,  -3,   5],
               [  2,   0,  20],
               [ -4,   0, 120]], dtype=int32)

A.cumprod(axis=1)
Out[21]:
        array([[  1,  -3, -15],
               [  2,   0,   0],
               [ -2, -16, -96]], dtype=int32)

A.trace()
Out[22]: 7
```

```
np.trace(A)
Out[23]: 7

np.diag(A).sum()
Out[24]: 7
```

5. Теоретико-множинні операції

Для теоретико-множинних операцій у numpy містяться наступні функції:

- **np.unique**(a, return_index=False, return_inverse=False, return_counts=False, axis=None) – знаходить унікальні елементи масиву і повертає їх в відсортованому масиві.
    - a – вхідний масив.
    - return_index – якщо True, то окрім самих унікальних елементів також будуть повертатися їх індекси у вхідному масиві. За замовчуванням return_index = False,
    - return_inverse – якщо True, то окрім самих унікальних елементів також будуть повертатися індекси унікального масиву, які можна використати для відновлення вхідного масиву. За замовчуванням return_inverse = False.
    - return_counts – якщо True, то окрім унікальних елементів також буде повертатися кількість входжень кожного з них у вхідному масиві. За замовчуванням return_counts = False.
    - axis – вказує вісь, за якою необхідно знайти унікальні елементи.
- **np.intersect1d**(ar1**,** ar2**,** assume_unique=False**,** return_indices=False)
    - ar1, ar2 – вхідні масиви.
    - assume_unique – якщо True, обидва вхідні масива вважаються унікальними.
    - return_indices – якщо True, повертаються індекси, відповідні перетину двох масивів.
- **np.union1d**(ar1**,** ar2) – повертає унікальний відсортований масив значень, які знаходяться в будь-якому з двох вхідних масивів.
    - a1, a2 – вхідні масиви
- **np.setdiff1d**(ar1**,** ar2**,** assume_unique=False) – повертає унікальні значення в ar1, яких немає в ar2.

- ar1 – вхідний масив.
- ar2 – вхідний масив порівняння.
- assume_unique – якщо True, обидва вхідних масиву вважаються унікальними, що може прискорити обчислення.

- **np.setxor1d(**ar1**,** ar2**,** assume_unique=False**)** – повертає відсортовані унікальні значення, які перебувають тільки в одному (а не в обох) вхідних масивах.
  - ar1, ar2 – вхідні масиви.
  - assume_unique – якщо True, обидва вхідних масиву вважаються унікальними, що може прискорити обчислення.

- **numpy.in1d**(ar1**,** ar2**,** assume_unique=False**,** invert=False) – повертає логічний масив тієї ж довжини, що і ar1, який має значення True, якщо елемент ar1 знаходиться в ar2, і False в іншому випадку.
  - ar1 - вхідний масив.
  - ar2 – значення, за якими буде перевірятися кожне значення ar1
  - assume_unique – якщо True, обидва вхідних масиву вважаються унікальними.
  - invert – якщо True, значення в повернутому масиві інвертуються (тобто False, якщо елемент ar1 знаходиться в ar2, і True в іншому випадку).

Приклад

```
A=np.array([1,2,3,1,5,2,1])

B=np.array([5,6,-1,2,0])

np.unique(A)
Out[1]: array([1, 2, 3, 5])

np.intersect1d(A,B)
Out[2]: array([2, 5])

np.union1d(A,B)
Out[3]: array([-1,  0,  1,  2,  3,  5,  6])

np.setdiff1d(A,B)
```

```
Out[4]: array([1, 3])

np.setxor1d(A,B)
Out[5]: array([-1,  0,  1,  3,  6])

np.in1d(A,B)
Out[6]: array([False,  True, False, False,  True,  True, False])
Out[7]: array([False,  True,  True, False, False], dtype=bool)

A[np.in1d(A,B)]
Out[8]: array([2, 5, 2])
```

6. Використання масивів в умовах

Функція **all ()** повертає True, якщо всі елементи задовольняють умові, істинні (або об'єкт порожній).

Функція **any ()** повертає True, якщо хоча б один елемент істинний, задовольняє якомусь умові.

Приклад

```
M=np.array([[1,4],[9,16]])

if (M>5).any():
   print("at least one element in M is larger than 5")
else:
   print("no element in M is larger than 5")
Out[1]: at least one element in M is larger than 5

if (M>5).all():
   print("all elements in M are larger than 5")
else:
   print("at least one element in M is not larger than 5")
Out[2]: at least one element in M is not larger than 5
```

7. Кусково-визначені функції

Метод **np.piecewise(x, condlist, funclist)** застосовує функцію з funclist до тієї частини масиву x, для якої виконується відповідна умова з condlist:

|--

```
        |funclist[0](x[condlist[0]])
  out = |funclist[1](x[condlist[1]])
        |...
        |funclist[n2](x[condlist[n2]])
        |--
```

Також можна вказати додаткові настройки функцій.

Приклад 1

```
x=np.arange(-3,4)

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

np.piecewise(x, [x<0, x>=0], [-1, 1])
Out[2]: array([-1, -1, -1,  1,  1,  1,  1])

np.piecewise(x, [x<0, x>=0], [lambda x:-x, lambda x:x])
Out[3]: array([3, 2, 1, 0, 1, 2, 3])
```

$$f(x,a)=\begin{cases} x/a, x<-1, \\ x^a, -1\le x<2, \\ ax, x\ge 2 \end{cases}$$

```
np.piecewise(x, [x<-1, (x>=-1)&(x<2), x>=2], [lambda x,a:x/a, lambda x,a:x**a, lambda x,a:x*a], a=2)
Out[4]: array([-1, -1,  1,  0,  1,  4,  6])
```

Застосування функції np.where є більш швидким порівняно з використанням np.piecewise.

Приклад 2

```
%timeit np.piecewise(x, [x<-1, (x>=-1)&(x<2), x>=2], [lambda x,a:x/a, lambda x,a:x**a, lambda x,a:x*a], a=2)
60.8 µs ± 1.22 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

%timeit a=2;np.where(x<-1,x/a,np.where((x>=-1)&(x<2),x**a,x*a))
25.5 µs ± 1.51 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

%timeit a=2;np.select([x<-1, (x>=-1)&(x<2), x>=2],[x/a,x**a,x*a])
127 µs ± 9.46 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

8. Застосування функцій уздовж осі

Метод **apply_along_axis()** застосовує функцію, яка подається, уздовж 1D зрізів вхідного масиву, причому зрізи беруться уздовж зазначеної вами осі.

Приклад

```
def avg_first_last(x):
   return (x[0]+x[-1])/2

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

np.apply_along_axis(avg_first_last, 0, A)
Out[1]: array([3., 4.])

np.apply_along_axis(avg_first_last, 1, A)
Out[2]: array([1.5, 3.5, 5.5])
```

9. Векторизация функцій

Використання for-циклів є більш повільним у порівнянні з використанням векторизованних функцій.

а. Векторизация функцій по одному аргументу

Скалярна функції Хевісайда повертає 1, якщо дане число більше або дорівнює нулю, і 0 – якщо менше нуля. Скалярная функція не працює з векторними вхідними параметрами.

Метод **numpy.vectorize** бере функцію f: a -> b і перетворює її в g: a [] -> b [].

Векторна функції Хевісайда приймає на вхід масив та перевіряє його, чи більше він 0. Працює з векторними і зі скалярними даними.

Векторизовані функції працюють швидше, ніж for-цикли, але тільки для масивів великої довжини. Порівняння:

```
%timeit Theta_vec(x)
17.5 µs ± 41.2 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit [Theta(i) for i in x]
2.61 µs ± 4.56 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

x=np.linspace(0,5,100)

%timeit Theta_vec(x)
```

32.5 µs ± 49.4 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

%timeit [Theta(i) for i in x]
31 µs ± 90.9 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

x=np.linspace(0,10,**1000**)

%timeit Theta_vec(x)
182 µs ± 691 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

%timeit [Theta(i) for i in x]
302 µs ± 1.93 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

Приклад

```
def Theta(x):
    if x>=0:
        return 1
    else:
        return 0

Theta(3)
Out[1]: 1

Theta(np.arange(-3,4))
Traceback (most recent call last):
  File "<ipython-input-25-00f2dabb5504>", line 1, in <module>
    Theta(np.arange(-3,4))
  File "<ipython-input-24-490e0ae603a8>", line 3, in Theta
    if x>=0:
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()

Theta_vec=np.vectorize(Theta)
Theta_vec(np.arange(-3,4))
Out[2]: array([0, 0, 0, 1, 1, 1, 1])

def Theta(x):
    return 1*(x>=0)
```

```
Theta(np.arange(-3,4))
Out[3]: array([0, 0, 0, 1, 1, 1, 1])

Theta(-5.5), Theta(2.6)
Out[4]: (0, 1)
```

b. Векторизация за частиною аргументів

Створюємо функцію, яка приймає 2 аргумента, один з яких – список:

```
def mypolyval(p, x):
    _p = list(p)
    res = _p.pop(0)
    while _p:
        res = res*x + _p.pop(0)
    return res
```

Векторізуємо її, за винятком аргументу p

```
vpolyval=np.vectorize(mypolyval, excluded=['p'])
```

Приклад

```
vpolyval(p=[1, 2, 3], x=[0, 1])
Out[1]: array([3, 6])
```

c. Векторизация функцій по 2 і більше аргументам

Функція **meshgrid ()** створює список масивів координатних сіток *N*-мірного координатного простору для зазначених одновимірних масивів координатних векторів. Координатний простір – це простір *N*-мірних точок-координат, причому кожній точці в такому просторі відповідає комбінація одного значення з кожного координатного масиву.

Застосування векторизованних функції до векторів призводить до помилки.

Аналогічно можна зробити векторизацію по 3 і більше аргументів. У результаті получимо 3-мірну матрицю.

Приклад

```
def f(x, y):
    return x**2+y**2

x=np.linspace(1, 2, 4)
```

```
y=np.linspace(3, 5, 6)
xx, yy=np.meshgrid(x, y)
print('x:\n', xx, '\ny:\n', yy)
F=np.vectorize(f)
print('F:\n', F(xx, yy))
 [[ 1.          1.33333333  1.66666667  2.        ]
    [ 1.        1.33333333  1.66666667  2.        ]
    [ 1.        1.33333333  1.66666667  2.        ]
    [ 1.        1.33333333  1.66666667  2.        ]
    [ 1.        1.33333333  1.66666667  2.        ]
    [ 1.            1.33333333  1.66666667  2.
   ]]
y:
    [[ 3.   3.   3.   3. ]
    [ 3.4  3.4  3.4  3.4]
    [ 3.8  3.8  3.8  3.8]
    [ 4.2  4.2  4.2  4.2]
    [ 4.6  4.6  4.6  4.6]
    [ 5.   5.   5.   5. ]]
F:
    [[ 10.          10.77777778  11.77777778  13.
   ]
    [ 12.56        13.33777778  14.33777778  15.56
   ]
    [ 15.44        16.21777778  17.21777778  18.44
   ]
    [ 18.64        19.41777778  20.41777778  21.64
   ]
    [ 22.16        22.93777778  23.93777778  25.16
   ]
    [ 26.          26.77777778  27.77777778  29.
   ]]

F(x, y)
Traceback (most recent call last):

  File "<ipython-input-42-55355a5ae9c4>", line 1, in <module>
    F(x,y)
```

```
  File "C:\ProgramData\Anaconda3\lib\site-packages\numpy\lib\function_base.py", line 2218, in __call__
    return self._vectorize_call(func=func, args=vargs)

  File "C:\ProgramData\Anaconda3\lib\site-packages\numpy\lib\function_base.py", line 2287, in _vectorize_call
    outputs = ufunc(*inputs)

ValueError: operands could not be broadcast together with shapes (4,) (6,)
```

```
def f(x, y, z):
    return x**2+y**2+z**2

x=np.linspace(1, 2, 4)
y=np.linspace(3, 5, 6)
z=np.linspace(-5,8,10)
xx, yy, zz=np.meshgrid(x, y, z)
F=np.vectorize(f)
Fxyz=F(xx, yy, zz)
```

10. Методи примірника u-функцій

Будь-яка бінарна **u**-функція в NumPy має спеціальні методи для виконання деяких векторних операцій.

Універсальна функція (або **u**-функція) – це функція, яка оперує з **ndarray** поелементно, підтримує укладання, приведення типів і ін.

a. Reduce

Метод **reduce(x)** агрегує значення масиву шляхом послідовного застосування бінарної операції. Початкове значення залежить від **u**-функції. Цей метод виконує послідовне додавання елементів масиву, початкове значення = 0

Порівняння методів **add.reduce** і **sum**. Метод add.reduce працює трохи швидше методу sum.

```
%timeit np.add.reduce(x)
1.56 µs ± 6.38 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
%timeit x.sum()
1.98 µs ± 4.21 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Виконує послідовне додавання елементів масиву A по кожному стовпчику.

np.add.reduce(A, **axis=0**)

Виконує послідовне додавання елементів масиву по кожному рядку.

np.add.reduce(A, **axis=1**)

Приклад

```
x=np.arange(10)
np.add.reduce(x)
Out[1]: 45

x.sum()
Out[2]: 45

A=np.random.randn(3,5)
A
Out[3]:
    array([[-0.7925988 , -0.69067494, -1.22537694,  1.2749111 ,
    0.62478216],
        [ 0.04341535,  0.07443779, -0.77868913, -0.62907152, -
    1.28563431],
        [ 0.06558344,  0.81569893, -0.22455737,  1.41195459,
    0.13544421]])

np.add.reduce(A, axis=0)
Out[4]: array([-0.68360001,  0.19946177, -2.22862344,  2.05779417, -
0.52540793])

np.add.reduce(A, axis=1)
Out[5]: array([-0.80895742, -2.57554182,  2.2041238 ])
```

b. Accumulate

Метод **accumulate(x)** агрегує значення масиву, зберігаючи всі проміжні результати. Породжує масив того ж розміру, що й вихідний, що містить проміжні «акумульовані» значення.

Функція **cumsum ()** повертає кумулятивну (накапливаемую) суму елементів масиву, в тому числі і по заданій осі (осях).

Порівняння методів **accumulate**(x) і **cumsum ().** Метод add.accumulate працює трохи швидше методу cumsum.

```
%timeit np.add.accumulate(x)
```

375 ns ± 1.03 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
%timeit x.cumsum()
623 ns ± 2.26 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

Приклад

```
x=np.arange(10)
np.add.accumulate(x)
Out[1]: array([ 0,  1,  3,  6, 10, 15, 21, 28, 36, 45], dtype=int32)

x.cumsum()
Out[2]: array([ 0,  1,  3,  6, 10, 15, 21, 28, 36, 45], dtype=int32)

A=np.random.randn(3,5)
A
Out[3]:
    array([[-0.7925988 , -0.69067494, -1.22537694,  1.2749111 ,
    0.62478216],
        [ 0.04341535,  0.07443779, -0.77868913, -0.62907152, -
    1.28563431],
        [ 0.06558344,  0.81569893, -0.22455737,  1.41195459,
    0.13544421]])

np.add.accumulate(A, axis=0)
Out[4]:
    array([[-0.7925988 , -0.69067494, -1.22537694,  1.2749111 ,
    0.62478216],
        [-0.74918345, -0.61623716, -2.00406607,  0.64583958, -
    0.66085214],
        [-0.68360001,  0.19946177, -2.22862344,  2.05779417, -
    0.52540793]])

np.add.accumulate(A, axis=1)
Out[5]:
    array([[-0.7925988 , -1.48327374, -2.70865068, -1.43373959, -
    0.80895742],
        [ 0.04341535,  0.11785313, -0.660836  , -1.28990751, -
    2.57554182],
        [ 0.06558344, 0.88128237, 0.656725 , 2.06867959, 2.2041238
    ]])
```

c. Outer

Метод **outer(x, y)** застосовує операцію до всіх парах елементів x і y. Результуючий масив має форму x.shape + y.shape. **Shape** повертає кортеж з кожним індексом, що має кількість відповідних елементів.

Метод **frompyfunc**(func, nin, nout) створює свою u-функцю, приймає вихідну функцію, кількість вхідних і вихідних аргументів, де nin – число вхідних параметрів, nout – число вихідних параметрів.

Метод outer працює швидше, ніж векторизованних функція F. Однак метод outer працює тільки для 2 аргументів.

```
%timeit F(xx, yy)
32 µs ± 27 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
%timeit f1.outer(x, y)
13.4 µs ± 67.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Приклад

```
def f(x, y):
   return x**2+y**2

x=np.linspace(1, 2, 4)
y=np.linspace(3, 5, 6)

f1=np.frompyfunc(f, 2, 1)

print(f1.outer(x, y))
[[10.0 12.559999999999999 15.44 18.64 22.159999999999997 26.0]
 [10.777777777777779 13.337777777777777 16.217777777777776
  19.417777777777778 22.937777777777775 26.777777777777778]
 [11.777777777777777 14.337777777777776 17.217777777777776
  20.417777777777778 23.937777777777775 27.777777777777778]
 [13.0 15.559999999999999 18.439999999999998 21.64
25.159999999999997 29.0]]
```

d. Reduceat

Метод **reduceat(x, bins)** редукує сусідні зрізи даних і породжує масив агрегатів, виконує «локальну редукцію». Приймає «межі інтервалів», що описують, як розбивати і агрегувати значення. Повертає локальні суми по зрізах.

Повертає локальні суми по зрізах A [:, 1: 3], A[:,3:]:

```
np.add.reduceat(A, [1,3], axis=1).
```

Приклад

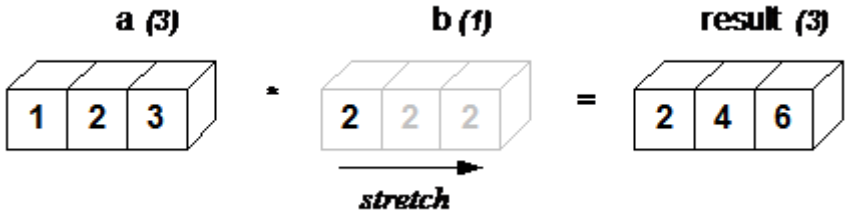
```
x=np.arange(10)
x
Out[1]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

np.add.reduceat(x, [1,5,8])
Out[2]: array([10, 18, 17], dtype=int32)

A=np.random.randn(3,5)
A
Out[3]:
    array([[-0.7925988 , -0.69067494, -1.22537694,  1.2749111 ,
    0.62478216],
        [ 0.04341535,  0.07443779, -0.77868913, -0.62907152, -
    1.28563431],
        [ 0.06558344,  0.81569893, -0.22455737,  1.41195459,
    0.13544421]])

np.add.reduceat(A, [1,3], axis=1)
Out[4]:
    array([[-1.91605188,  1.89969326],
        [-0.70425134, -1.91470582],
        [ 0.59114156,  1.5473988 ]])
```

11. Укладання (broadcasting)

Правило укладання: 2 масиви сумісні по укладенню, якщо для обох останніх вимірювань (тобто відлік яких ведеться з кінця) довжини осей збігаються або хоча б одна довжина дорівнює 1. Тоді укладання проводиться по відсутнім вимірам або за вимірюваннями довжини 1.

Бродкастінг (broadcasting) – автоматичне розширення розмірності (ndim) і розмірів (shape) масивів, при здійсненні операцій (додавання, множення і подібні) над масивами з різними розмірами або размерностями за умови, що вони сумісні з правилами бродкастінг.

Наприклад, нам потрібно помножити вектор x = np.array([1, 2, 3]) на скаляр 2. NumPy не примушує нас вручну перетворювати скаляр (2) в вектор [2, 2, 2], він сам додає розмірність і клонує вміст потрібне число раз, а потім вже виробляє поелементне множення.



Бродкастінг в даному випадку не тільки дає зручність, але і приріст продуктивності і економію пам'яті, тому що NumPy робить ці операції ефективно у своєму оптимизированном Сі-коді і не створює зайвих копій даних.

При виконанні операцій між матрицями та векторами **укладаються строки**: (m,n), (1,n) -> (m,n)

1	-3	5
2	0	4
-2	8	6

\+

1	2	3

=

2	-1	8
3	2	7
-1	10	9

Укладання 1-мірного масиву по вісі 0

При виконанні операцій між матрицями та вектор-стовбцями **укладаються стовбці**: (m,n), (m,1) -> (m,n)

1	-3	5
2	0	4
-2	8	6

\+

1
2
3

=

2	-2	6
4	2	6
1	11	9

Укладання 1-мірного масиву по вісі 1

Приклад

```
A=np.array([[1, -3, 5], [2, 0, 4], [-2, 8, 6]])
A
Out[1]:
        array([[ 1, -3,  5],
               [ 2,  0,  4],
               [-2,  8,  6]])
x=np.array([1, 2, 3])

A + x
Out[2]:
```

```
        array([[ 2, -1,  8],
               [ 3,  2,  7],
               [-1, 10,  9]])

A - x
Out[3]:
        array([[ 0, -5,  2],
               [ 1, -2,  1],
               [-3,  6,  3]])

A * x
Out[4]:
        array([[ 1, -6, 15],
               [ 2,  0, 12],
               [-2, 16, 18]])

A**x
Out[5]:
        array([[  1,   9, 125],
               [  2,   0,  64],
               [ -2,  64, 216]], dtype=int32)

x2=np.array([[1],[2],[3]])

A+x2
Out[6]:
        array([[ 2, -2,  6],[ 4,  2,  6], [ 1, 11,  9]])

A[x<3]
Out[7]:
        array([[ 1, -3,  5], [ 2,  0,  4]])
```

12. Завдання та вправи

1. Які результати отримаємо після виклику наступних функцій:

```
A
array([[ 1, -3,  5],
       [ 8,  2,  4],
       [ 3,  9,  0]])
```

- A[A.argmax(axis=0)]
- A[A.argmax(axis=1)]
- A[A.argmax()]

2. За допомогою яких теоретично-множинних операцій можна отримати наступні значення:

A
array([1, -3, 0, 6, 2, 1, -1, 2])
B
array([3, 2, 1, 4, -1, 1, 3])

- array([-3, -1, 0, 1, 2, 6])
- array([-1, 1, 2, 3, 4])
- array([-1, 1, 2])
- array([-3, -1, 0, 1, 2, 3, 4, 6])
- array([-3, 0, 6])
- array([3, 4])
- array([-3, 0, 3, 4, 6])

3. Застосувати функцію my_median для обчислення медіани по стовбцям та строкам матриці A. Порівняти з результатом роботи стандартної функції.

```
def my_median(x):
    y=np.sort(x)
    n=len(y)
    if n%2==1:
        return y[n//2]
    else:
        return (y[n//2-1]+y[n//2])/2

A=np.random.randint(-5,10,(4,5))
A
Out[113]:
array([[-1,  6,  3,  3,  4],
       [ 0, -5,  1,  8,  6],
       [-3,  7,  6,  4,  3],
       [ 7,  9,  4, -2, -2]])
```

4. Перетворити масив строк у нижньому регістрі в масив строк, що починаються з великої букви.

5. Перетворити масив строк в нижньому регістрі в масив речень (строк, що починаються з великої букви і закінчуються крапкою).

6. Визначити, які результати отримаємо у результаті складання матриці A з векторами x1 та x2.

```
A=np.matrix("2,-1,0;3,5,9;-8,7,11")
A
Out[19]:
matrix([[ 2, -1,  0],
        [ 3,  5,  9],
        [-8,  7, 11]])
x1=np.matrix('1,1,1')
x2=np.matrix('2;2;2')
```

- A+x1
- A+x2

VI Поліноми

1. Створення poly1d

- **poly1d**([list]) – створення полінома за його коефіцієнтами, починаючи від старшого степеня
- **variable**='z' – таким чином можна вказати інше позначення для змінної
- **r**=True – за такої умови будуватиметься багаточен із заданим корінням
- Нехай p – поліном. Отримати значення багаточена в точці x = 1 можна наступним чином – p(1).
- p.**c –** таким чином можна отримати коефіцієнти полінома
- p.**r** або np.**roots**(p) **–** таким чином можна отримати коріння багаточена (np.roots(p.c) працює не тільки з об'єктом poly1d, але і з коефіцієнтами полінома, представленими у вигляді ndarray)
- p.**order –** таким чином можна отримати порядок полінома
- p[1] – так можна отримати коефіцієнт при 1-му степені x

Приклад

```
p=np.poly1d([1,2,3])
print(p)
    2
1 x + 2 x + 3
p=np.poly1d([1,2,3], variable='z')
print(p)
    2
1 z + 2 z + 3
```

```
p=np.poly1d([1,2,3], r=True)
print(p)
    3      2
1 x - 6 x + 11 x - 6
```

```
p(1)
Out[1]: 0.0

p(5)
Out[2]: 24.0

p.c
Out[3]: array([ 1., -6., 11., -6.])

p.r
Out[4]: array([3., 2., 1.])

np.roots(p)
Out[5]: array([3., 2., 1.])

np.roots(p.c)
Out[6]: array([3., 2., 1.])

p.order
Out[7]: 3

p[1]
Out[8]: 11.0

p[0]
Out[9]: -6.0

p[3]
Out[10]: 1.0
```

2. Створення Polynomial

```
import numpy.polynomial.polynomial as P
```

- p = P.**Polynomial**([2, 1]) – таким чином створюється поліном виду 2 + x, коефіцієнти при степенях вказуються в порядку від найменшої до найбільшої поспіль: $a_0+a_1*x+a_2*x^2+\ldots a_n*x^n$
- p.**degree**() – таким чином можна отримати степень багаточена
- p.**roots**() – таким чином можна отримати коріння багаточена

Приклад

```
p=P.Polynomial([2, 1])

p
Out[1]: Polynomial([ 2.,  1.], [-1,  1], [-1,  1])

print(p)
poly([ 2.  1.])

p(-2)
Out[2]: 0.0

p(1)
Out[3]: 3.0

p.degree()
Out[4]: 1

p.roots()
Out[5]: array([-2.])
```

3. Арифметичні операції з poly1d

Розглянемо арифметичні операції з poly1d.

- “ + ” – додавання
- “ - ” – віднімання
- “ * ” – множення
- “ / ” – ділення (повертає ділене і залишок)
- “ ** ” – піднесення до степеня
- np.**square**(p1) – підносить до квадрату коефіцієнти багаточена

Приклад

```
p1=np.poly1d([1,0,2,3])
print(p1)
      3
```

```
1 x + 2 x + 3
```

```
p2=np.poly1d([0,5,4,-1,2])
print(p2)
    3      2
5 x + 4 x - 1 x + 2
```

```
p1+p2
Out[1]: poly1d([6, 4, 1, 5])
```

```
p1-p2
Out[2]: poly1d([-4, -4,  3,  1])
```

```
p1*p2
Out[3]: poly1d([ 5,  4,  9, 25, 10,  1,  6])
```

```
p1/p2
Out[4]: (poly1d([0.2]), poly1d([-0.8,  2.2,  2.6]))
```

```
p1+5
Out[5]: poly1d([1, 0, 2, 8])
```

```
p1*2
Out[6]: poly1d([2, 0, 4, 6])
```

```
p1**2
Out[7]: poly1d([ 1,  0,  4,  6,  4, 12,  9])
```

```
np.square(p1)
Out[8]: array([1, 0, 4, 9], dtype=int32)
```

4. Арифметичні операції з Polynomial

Приклад

```
p1=P.Polynomial([1, 0, 2, 3]) #1 + 2 * x**2 + 3 * x**3
p2=P.Polynomial([0, 5, 4, -1, 2]) #5 * x + 4 * x**2 - x**3 + 2 * x**4
```

```
p1+p2 #1 + 5 * x + 6 * x**2 + 2 * x**3 + 2 * x**4
Out[1]: Polynomial([ 1.,  5.,  6.,  2.,  2.], [-1.,  1.], [-1.,  1.])
```

p1-p2 #1 – 5 * x – 2 * x^2 + 4 * x^3 – 2 * x^4
Out[2]: Polynomial([1., -5., -2., 4., -2.], [-1., 1.], [-1., 1.])

p1*p2 #5 * x + 4 * x^2 + 9 * x^3 + 25 * x^4 + 10 * x^5 + x^6 + 6 * x^7
Out[3]: Polynomial([0., 5., 4., 9., 25., 10., 1., 6.], [-1., 1.], [-1., 1.])

p1/5
Out[4]: Polynomial([0.2, 0. , 0.4, 0.6], [-1., 1.], [-1., 1.])

p1**2
Out[5]: Polynomial([1., 0., 4., 6., 4., 12., 9.], [-1., 1.], [-1., 1.])

5. Диференціювання

Зауважимо, що коефіцієнти багаточена можна задати як списком, так і кортежем.

- P.**polyder**(c) – таким чином можна отримати першу похідну
- P.polyder(c, 3) – так можемо отримати похідну іншого порядку, у даному випадку третю
- P.polyder(c, scl=-1) – таким чином можна отримати (d/d(-x))(c)
- P.polyder(c, 2, -1) – таким чином можна отримати (d**2/d(-x)**2)(c)

P.polyder(p) працює також для об'єктів poly1d.

Приклад

c = (1, 2, 3, 4) #1 + 2x + 3x**2 + 4x**3

P.**polyder**(c) #2 + 6x + 12x**2
Out[1]: array([2., 6., 12.])

P.polyder(c, 3)
Out[2]: array([24.])

P.polyder(c, scl=-1) #-2 – 6x – 12x**2
Out[3]: array([-2., -6., -12.])

P.polyder(c, 2, -1) #6 + 24x
Out[4]: array([6., 24.])

```
p=np.poly1d([1,2,3,4])
P.polyder(p)
Out[5]: array([ 2.,  6., 12.])
```

6. Інтегрування

- P.**polyint**(c) – таким чином можна проінтегрувати багаточен
- P.polyint(c, 2) – таким чином можемо проінтегрувати багаточен кілька разів, у цьому випадку двічі

Зауважимо, що P.polyint(p) працює також для об'єктів poly1d.

Приклад

```
c = (1, 2, 3, 4) #1 + 2x + 3x**2 + 4x**3

P.polyint(c)
Out[1]: array([ 0.,  1.,  1.,  1.,  1.])

P.polyint(c, 2)
Out[2]:
array([ 0.,   0. ,  0.5,  0.33333333,  0.25,  0.2 ])

P.polyint(p)
Out[3]: array([0., 1., 1., 1., 1.])
```

7. Наближення функцій поліномами

- P.**polyfit**(x, y, 3) – знаходить поліном потрібного степеня (цьму випадку степеня 3), що наближає функцію y(x) за її значеннями в точках методом найменших квадратів
- c, stats=P.polyfit(x, y, 3, **full**=True) – так окрім коефіцієнтів поліном виводитиме додаткову інформацію про SVD-розкладання, яке використовувалося
- yLSF=P.**polyval**(x, c) – повертає значення поліному, що визначається коефіцієнтами c, в точках x

Приклад

```
x=np.linspace(1,2,30)
y=x**3+x

c=P.polyfit(x,y,3)
c
```

```
Out[1]: array([1.04626664e-14, 1.00000000e+00, 5.17692984e-15,
1.00000000e+00])

c, stats=P.polyfit(x, y, 3, full=True)
c
Out[2]: array([1.04626664e-14, 1.00000000e+00, 5.17692984e-15,
1.00000000e+00])

stats
Out[3]:
[array([7.56986125e-30]),
 4,
 array([1.96422545e+00, 3.75280650e-01, 3.13234034e-02, 1.28997727e-
03]),
 6.661338147750939e-15]

yLSF=P.polyval(x, c)
yLSF
Out[4]:
        array([ 2. ,  2.14153922,  2.29045881,  2.4470048 ,  2.61142318,
                2.78395998,  2.96486121,  3.15437287,  3.35274099,
        3.56021157,
                3.77703063,  4.00344418,  4.23969822,  4.48603879,
        4.74271188,
                5.00996351,  5.28803969,  5.57718644,  5.87764976,
        6.18967567,
                6.51351019,  6.84939932,  7.19758908,  7.55832547,
        7.93185452,
                8.31842224,  8.71827463,  9.13165771,  9.5588175 , 10. ])

y
Out[5]:
        array([ 2. ,  2.14153922,  2.29045881,  2.4470048 ,  2.61142318,
                2.78395998,  2.96486121,  3.15437287,  3.35274099,
        3.56021157,
                3.77703063,  4.00344418,  4.23969822,  4.48603879,
        4.74271188,
```

```
        5.00996351,      5.28803969,      5.57718644,      5.87764976,
    6.18967567,
        6.51351019,      6.84939932,      7.19758908,      7.55832547,
    7.93185452,
        8.31842224,  8.71827463,  9.13165771,  9.5588175 , 10.])

y=x**3+x+np.random.rand(len(x))

c, stats=P.polyfit(x,y,3,full=True)
c
Out[6]: array([-6.38147806, 15.33448948, -9.5133534 ,  3.01059318])

stats
Out[7]:
    [array([2.01112213]), 4,
     array([1.96422545e+00,      3.75280650e-01,      3.13234034e-02,
    1.28997727e-03]),
     6.661338147750939e-15]

yLSF=P.polyval(x,c)
yLSF
Out[8]:
    array([  2.60219579,    2.63614634,    2.69845146,    2.78782905,
    2.90299702,
        3.04267327,    3.20557572,    3.39042226,    3.5959308 ,
    3.82081924,
        4.0638055 ,    4.32360747,    4.59894306,    4.88853018,
    5.19108674,
        5.50533063,    5.82997976,    6.16375204,    6.50536538,
    6.85353767,
        7.20698683,    7.56443076,    7.92458736,    8.28617454,
    8.64791021,
        9.00851227,    9.36669862,    9.72118718,   10.07069584,
    10.41394252])
Out[9]:
    array([  2.63583773,    3.03924265,    2.50509063,    2.5531043 ,
    2.67148564,
```

2.87889284, 3.18887579, 3.35114976, 3.48832843, 4.48066887,
4.13250967, 4.24616613, 4.70031962, 5.25318362, 5.01322169,
5.40544698, 5.66615485, 6.22039708, 6.39041039, 7.05784029,
6.54671117, 7.53865334, 7.94960509, 8.27111402, 8.87527279,
9.23884136, 9.4085156 , 9.92583614, 9.90081186, 10.28441712])

8. Завдання та вправи

1. Обчислити наступні вирази:
   - (x^2-5*x-4)/(x-1)
   - (x^2+10*x-20)^2
   - (x^3+3*x)*(x^2-7*x+10)
2. Обчислити наступні похідні:
   - d/dx(15*x^2-10*x+8)
   - d2/dx2(9*x^3+15*x^2-6*x+12)
3. Обчислити первісні від наступних поліномів:
   - x^5-12*x^3+8
   - x^3-2*x^2+11*x
   - 4*x^3-3*x^2+2*x-1

VII Зміна форми

1. Зміна форми (reshaping)

Функція перетворення в вектор-рядок, копіювання даних не проводиться:

np.**reshape**(a, newshape)

Тут:

- **a** – масив, форму якого необхідно змінити;
- **newshape** – визначає форму вихідного масиву.

Зазначений параметр повинен бути сумісний з формою вихідного масиву. Ціле число стискає вихідний масив до однієї осі. Одне з вимірів може дорівнювати -1, що призводить до автоматичного обчислення довжини осі.

Приклад

G=np.random.randn(10,1)

```
Out[1]:
array([[-0.25058183],
       [ 0.38224959],
       [ 0.79618226],
       [-1.67809708],
       [ 1.44634553],
       [ 1.53027806],
       [ 1.0533384 ],
       [ 0.08403082],
       [ 0.59833224],
       [-0.59063064]])

G = G.reshape((1, 9))

Out[2]:
array([[ 0.43835902,  0.47291798,  0.86477438,  0.30105818,  0.31315592,
        0.45543947,  0.03766296,  0.90630304,  0.63098337]])

G = G.reshape((2, 5))

Out[3]:
array([[-0.03632487,  -0.18830902,  -0.00926936,  1.45158077,  -
0.48706734],
       [-0.64909743,  -0.95648782,  -0.0297074 ,  -0.66033575,  -
        0.97078781]])
```

Якщо другий аргумент в кортежі дорівнює -1, то кількість стовпців результуючої матриці обчислюється:

Приклад

```
G.reshape((2, -1))

Out[1]:
array([[-0.25058183,  0.38224959,  0.79618226,  -
1.67809708,  1.44634553],
       [ 1.53027806,  1.0533384 ,  0.08403082,  0.59833224, -0.59063064]])
```

Функція перетворення в одновимірний масив:

a**.flatten**()
Тут: **a** – вхідний масив.

Приклад

```
G = G.flatten()
G
Out[1]:
array([ 0.43835902,  0.47291798,  0.86477438,  0.30105818,  0.31315592,
        0.45543947,  0.03766296,  0.90630304,  0.63098337])
```

Функція прибирає розмірності, що дорівнюють 1 (в даному випадку перетворило вектор-стовпець в «плоский» масив):
np**.squeeze**(a)
Тут: **a** – масив в якому необхідно видалити вісь довгою 1.

Приклад

```
G=np.random.randn(10,1)
G
Out[1]:
array([[-0.23826621],
       [-0.01544928],
       [ 0.15698403],
       [-0.67359514],
       [-0.25781656],
       [ 1.98844669],
       [ 0.06897684],
       [-0.84949844],
       [-0.437917  ],
       [ 2.00550166]])

np.squeeze(G)
Out[2]:
array([-0.23826621, -0.01544928,  0.15698403, -0.67359514, -0.25781656,
        1.98844669,  0.06897684, -0.84949844, -0.437917  ,  2.00550166])
```

Функція змінює сам масив A, аналогічно A = A.reshape ((2, 2)):
np**.resize**(a, new_shape)
Тут: **a** – масив NumPy або будь-який об'єкт який може бути перетворений в масив NumPy; **new_shape** – визначає форму вихідного масиву.

Якщо зазначені розміри для вихідного масиву більше розмірів вихідного, то новий масив заповнюється повторюваними копіями вихідного масиву. Така поведінка відрізняється від функції a.resize (new_shape), яка заповнює відсутні елементи нулями.

Приклад

```
A=np.arange(0, 4)
A
Out[1]: array([0, 1, 2, 3])

A.resize((2, 2))
A
Out[2]:
array([[0, 1],
       [2, 3]])
```

Метод resize працює швидше, ніж reshape.

Приклад

```
%timeit A.reshape((2,2))
289 ns ± 2.08 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

%timeit A.resize((2,2))
208 ns ± 1.98 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Функція повертає одновимірний масив з елементів масиву A, завжди створює копію даних.

Приклад

```
A.flatten()
Out[1]: array([0, 1, 2, 3])
```

Функція повертає одновимірний масив з елементів масиву A в разі, якщо A є одновимірним масивом, повертає посилання на нього:

np.**ravel**(a)

Тут: **a** – масив будь-якої форми і розмірності, який буде стиснутий до однієї осі.

Приклад

```
A.ravel()
Out[1]: array([0, 1, 2, 3])
```

Метод ravel працює набагато швидше, ніж flatten.

Приклад

```
%timeit A.flatten()
707 ns ± 0.619 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

%timeit A.ravel()
172 ns ± 0.0474 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Функція лінеарізації по рядках (по рядках представляються масиви в C), використовується за умовчанням.

Приклад

```
A.ravel('C')
Out[1]: array([0, 1, 2, 3])
```

Функція лінеарізації за стовпцями (по стовпцях подаються масиви в Fortran):

Приклад

```
A.ravel('F')
Out[1]: array([0, 2, 1, 3])
```

a. Ndenumerate – N нумерація

Функція повертає ітератор, що видає пари координат і значень масиву.

numpy.**ndenumerate**(A)

Тут:

- **i** – кортеж індексів,
- **x** – відповідні значення, прохід до масиву **A** здійснюється через підрядник – по A.flat.

Приклад

```
A=np.arange(15)
A=A.reshape((3,5))
A
Out[1]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
for i, x in np.ndenumerate(A):
   print(i, x)

Out[2]:
(0, 0) 0
(0, 1) 1
(0, 2) 2
(0, 3) 3
(0, 4) 4
(1, 0) 5
(1, 1) 6
(1, 2) 7
(1, 3) 8
(1, 4) 9
(2, 0) 10
(2, 1) 11
(2, 2) 12
(2, 3) 13
(2, 4) 14
```

За допомогою функції enumerate можна отримати номери рядків матриці:

Вбудована в Python функція **enumerate()** застосовується для ітеріруемих колекцій (рядки, списки, словники та ін.) І створює об'єкт, який генерує кортежі, що складаються з двох елементів – індексу елемента і самого елемента.

np.enumerate(a)

Тут **a** – вхідний масив.

Приклад

```
for i,x in enumerate(A):
   print(i,x)
Out[1]
0 [0 1 2 3 4]
1 [5 6 7 8 9]
2 [10 11 12 13 14]
```

2. Додавання нового виміру

Перетворення у вектор-стовпець:

np.newaxis використовується для збільшення розміру існуючого масиву ще на один вимір, коли використовується один раз.

Приклад

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

np.shape(v)
Out[1]: (3,)

v1=v[:, np.newaxis]
v1.shape
Out[2]: (3, 1)

v1
Out[3]:
array([[1],
       [2],
       [3]])

v2=v[np.newaxis,:]
v2.shape
Out[4]: (1, 3)

v2
Out[5]: array([[1, 2, 3]])
```

Приклад Обчислити суму матриці A і вектора x за стовпцями.

```
A=np.arange(20).reshape(5,4)
A
Out[1]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])

x=np.arange(5)
x
Out[2]: array([0, 1, 2, 3, 4])
```

```
A+x
Traceback (most recent call last):

  File "<ipython-input-28-2388f1cfb3ce>", line 1, in <module>
    A+x

ValueError: operands could not be broadcast together with shapes (5,4) (5,)

x=x[:,np.newaxis]
x
Out[3]:
array([[0],
       [1],
       [2],
       [3],
       [4]])

A+x
Out[4]:
array([[ 0,  1,  2,  3],
       [ 5,  6,  7,  8],
       [10, 11, 12, 13],
       [15, 16, 17, 18],
       [20, 21, 22, 23]])
```

3. Перестановка осей

Звичайне транспонування: A.T.

Змінити або перестановити осі масиву; повертає змінений масив.

Для масиву a з двома осями транспонування (a) дає матрицю транспонування.

A.**transpose**(A)

Тут: **A** – вхідний масив.

Приклад

```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A
Out[1]:
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
```

```
        [20, 21, 22, 23, 24],
        [30, 31, 32, 33, 34],
        [40, 41, 42, 43, 44]])

Out[2]:
array([[ 0, 10, 20, 30, 40],
        [ 1, 11, 21, 31, 41],
        [ 2, 12, 22, 32, 42],
        [ 3, 13, 23, 33, 43],
        [ 4, 14, 24, 34, 44]])

A=np.arange(16).reshape(2,2,4)
A
Out[3]:
array([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7]],

         [[ 8,  9, 10, 11],
          [12, 13, 14, 15]]])

Переставляє осі за номерами:
A.transpose((2, 0, 1))
" Було: 0 –матриці, 1 – рядки, 2 – стовпці
Стало: 0 – стовпці, 1 – матриці, 2 – рядки "
Out[4]:
        array([[[ 0,  4],
                [ 8, 12]],

               [[ 1,  5],
                [ 9, 13]],

               [[ 2,  6],
                [10, 14]],

               [[ 3,  7],
                [11, 15]]])
```

Міняє місцями 2 конкретні осі:

A.**swapaxes**(axis1, axis2)

Тут: **A** – масив, осі якого повинні бути переміщені; **axis1** – перша вісь; **axis2** – друга вісь.

Приклад

```
A.swapaxes(0, 2)
Out[1]:
        array([[[ 0,  8],
                [ 4, 12]],

               [[ 1,  9],
                [ 5, 13]],

               [[ 2, 10],
                [ 6, 14]],

               [[ 3, 11],
                [ 7, 15]]])

A.swapaxes(0, 1)
Out[2]:
        array([[[ 0,  1,  2,  3],
                [ 8,  9, 10, 11]],

               [[ 4,  5,  6,  7],
                [12, 13, 14, 15]]])
```

«Відкочується» вісь 1 до осі 0, еквівалентно A.swapaxes (0, 1):

np.**rollaxis**(a, axis)

Тут: **a** – вхідний масив; **axis** – вісь, яку потрібно котити. Положення інших осей не змінюються відносно одна одної.

Приклад

```
np.rollaxis(A, 1)
Out[1]:
        array([[[ 0,  1,  2,  3],
                [ 8,  9, 10, 11]],

               [[ 4,  5,  6,  7],
                [12, 13, 14, 15]]])
```

«Відкочується» вісь 2 до осі 0, при цьому змінює 2 і 1, а потім 1 і 0, еквівалентно A.swapaxes (1,2) .swapaxes (0,1):

```
np.rollaxis(A, 2)
Out[2]:
        array([[[ 0,  4],
                [ 8, 12]],

               [[ 1,  5],
                [ 9, 13]],

               [[ 2,  6],
                [10, 14]],

               [[ 3,  7],
                [11, 15]]])
```

«Відкочується» вісь 2 до осі 1, еквівалентно A.swapaxes (2,1):

```
np.rollaxis(A,2,1)
Out[3]:
        array([[[ 0,  4],
                [ 1,  5],
                [ 2,  6],
                [ 3,  7]],

               [[ 8, 12],
                [ 9, 13],
                [10, 14],
                [11, 15]]])
```

4. Поворот

Поворот матриці на 90 градусів проти годинникової стрілки:

np.**rot90**(a)

Тут:

- a – масив NumPy або будь-який об'єкт який може бути перетворений в масив NumPy, при цьому вхідний масив повинен мати не менше двох вимірювань;

- k – (необов'язковий) визначає кількість поворотів, k = 1 (за замовчуванням) відповідає повороту на 90 градусів, k = 2 відповідає 180 градусам і т. д .;
- axes –- (необов'язковий) послідовність з двох цілих чисел - номерів осей які визначають площину обертання. Числа (осі) повинні бути різними. За замовчуванням axes = (0, 1)

Приклад

```
A=np.array([[1,2],[3,4]], dtype=int)
A
Out[1]:
array([[1, 2],
       [3, 4]])

np.rot90(A)
Out[2]:
array([[2, 4],
       [1, 3]])

np.rot90(A, 2)

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

np.rot90(A, 3)

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

np.rot90(A, 4)

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

A=np.arange(16).reshape(2,2,4)
A
```

```
Out[6]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
```

Якщо в матриці більше, ніж 2 осі, можна вказувати осі, щодо яких здійснювати поворот на 90 градусів.

Приклад

```
np.rot90(A, axes=(0,1))
Out[1]:
array([[[ 4,  5,  6,  7],
        [12, 13, 14, 15]],

       [[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]]])

np.rot90(A, axes=(1,2))
Out[2]:
array([[[ 3,  7],
        [ 2,  6],
        [ 1,  5],
        [ 0,  4]],

       [[11, 15],
        [10, 14],
        [ 9, 13],
        [ 8, 12]]])

np.rot90(A, k=2, axes=(1,2))
Out[3]:
array([[[ 7,  6,  5,  4],
        [ 3,  2,  1,  0]],

       [[15, 14, 13, 12],
        [11, 10,  9,  8]]])
```

5. Повторення елементів

Повторення елементів можна отримати за допомогою функцій:

1) np.**arange**([start,]stop, [step,], dtype=None)
   - start – (необов'язковий) початок, за замовчуванням start= 0;
   - stop – кінець (не включно);
   - step – (необов'язковий) крок, за замовчуванням step=1;
   - dtype – (необов'язковий) тип.

   Приклад

   x=np.arange(5)
   x
   Out[1]: array([0, 1, 2, 3, 4])

2) np.**repeat**(object[, times], axis)
   - object – об'єкт, який слід повертати повторно;
   - times – кількість повторень; якщо не вказано, то буде повертати нескінченно;
   - axis – вісь, уздовж якої слід повторювати значення.

   Приклад Повторити:
   - кожен елемент 2 рази;
   - кожен елемент відповідно з числами, зазначеними в масиві, розміри масивів повинні збігатися!
   - кожен елемент 3 рази, причому повертає одновимірний масив;
   - рядки;
   - стовпці;
   - 1-й рядок 2 рази, 2-й – 3 рази.

   x.**repeat**(2)
   Out[1]: array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4])

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

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

   np.repeat(a, 3)
   Out[3]: array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4])

```
a.repeat(2, axis=0)
Out[4]:
array([[1, 2],
       [1, 2],
       [3, 4],
       [3, 4]])

a.repeat(2, axis=1)
Out[5]:
       array([[1, 1, 2, 2],
              [3, 3, 4, 4]])

a.repeat([2,3], axis=0)
Out[6]:
       array([[1, 2],
              [1, 2],
              [3, 4],
              [3, 4],
              [3, 4]])

a.repeat([2,3], axis=1)
Out[7]:
array([[1, 1, 2, 2, 2],
       [3, 3, 4, 4, 4]])
```

3) np.**tile**(a, rep)

- a – будь-який об'єкт, який може бути перетворений в масив NumPy;
- rep – послідовність цілих чисел, які задають повторення масиву **a** уздовж кожної осі.

Приклад "Викласти плитками" матрицю:

- 3 рази;
- 1 раз по рядках і 2 по стовпцях;
- 2 рази по рядках і 1 по стовпцях;
- 3 рази по рядках і 2 по стовпцях.

```
np.tile(a, 3)
Out[1]:
       array([[1, 2, 1, 2, 1, 2],
```

```
            [3, 4, 3, 4, 3, 4]])

np.tile(a, (1,2))
Out[2]:
      array([[1, 2, 1, 2],
            [3, 4, 3, 4]])

np.tile(a, (2,1))
Out[3]:
      array([[1, 2],
            [3, 4],
            [1, 2],
            [3, 4]])

np.tile(a, (3,2))
Out[4]:
      array([[1, 2, 1, 2],
            [3, 4, 3, 4],
            [1, 2, 1, 2],
            [3, 4, 3, 4],
            [1, 2, 1, 2],
            [3, 4, 3, 4]])
```

6. Конкатенація масивів

Функції для конкатенації масивів мають наступну сигнатуру:

1) np.**concatenate**((a$_1$, a_2, ..., a$_n$), axis=0)
   - (a$_1$, a_2, ..., a$_n$) – будь-які об'єкти, які можуть бути перетворені в масиви NumPy;
   - axis – визначає вісь уздовж якої з'єднуються масиви, за замовчуванням axis = 0.

Приклад

«Склеює» масиви a і b по **вертикалі:**

```
a=np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=0)
Out[1]:
      array([[1, 2],
            [3, 4],
```

```
        [5, 6]])
```

«Склеює» масиви a і b по **горизонталі:**

```
np.concatenate((a, b.T), axis=1)
Out[2]:
    array([[1, 2, 5],
        [3, 4, 6]])
```

2) np.**vstack**$((a_1, a_2, ..., a_n))$ - «Склеює» масиви a і b по **вертикалі**

- $(a_1, a_2, ..., a_n)$ – будь-які об'єкти, які можуть бути перетворені в масиви NumPy.

Приклад

```
np.vstack((a, b))
Out[1]:
    array([[1, 2],
        [3, 4],
        [5, 6]])
```

3) np.**hstack**$((a_1, a_2, ..., a_n))$ – «Склеює» масиви a і b по **горизонталі**

- $(a_1, a_2, ..., a_n)$ – будь-які об'єкти, які можуть бути перетворені в масиви NumPy;

Приклад

```
np.hstack((a, b.T))
Out[1]:
    array([[1, 2, 5],
        [3, 4, 6]])
```

4) np.**row_stack**$((a_1, a_2, ..., a_n))$ – еквівалентно vstack.

- $(a_1, a_2, ..., a_n)$ – будь-які об'єкти, які можуть бути перетворені в масиви NumPy.

Приклад:.

```
np.row_stack((a,b))
Out[1]:
array([[1, 2],
    [3, 4],
    [5, 6]])
```

5) np.**coloumn_stack**$((a_1, a_2, ..., a_n))$ – еквівалентно hstack.

- $(a_1, a_2, ..., a_n)$ – будь-які об'єкти, які можуть бути перетворені в масиви NumPy.

Приклад

```
np.column_stack((a,b.T))
Out[1]:
array([[1, 2, 5],
       [3, 4, 6]])
```

Функція concatenate працює швидше функцій vstack і hstack.

Приклад

```
%timeit np.concatenate((a,b),axis=0)
1.79 µs ± 2.85 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

%timeit np.vstack((a,b))
4.96 µs ± 18.2 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit np.row_stack((a,b))
4.93 µs ± 11.5 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit np.concatenate((a,b.T),axis=1)
2.02 µs ± 9.35 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit np.hstack((a,b.T))
5.15 µs ± 13.3 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit np.column_stack((a,b.T))
3.7  µs ± 7.41 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

7. Розбиття масивів

Функції для розбиття масивів мають таку сигнатуру:

np.**split**(a, indices_or_sections, axis=0)

- a – Масив, який необхідно розбити на підмасиви;
- indices_or_sections – якщо цей параметр є одновимірним масивом відсортованих по зростанню цілих чисел, то вздовж зазначеної осі, масив буде розбитий на відповідні зазначеним числам проміжки;
- axis = 0 – визначає вісь уздовж якої відбувається розбиття масиву. За замовчуванням дорівнює 0.

np.**vsplit**(a, indices_or_sections) – розбиває масив по вертикалі;
np.**hsplit**(a, indices_or_sections) – розбиває масив по горизонталі.

Приклад Функції np.split, np.vsplit, np.hsplit розбивають масив:

- на 3 масива по 3 елементи, якщо довжина масива не дорівнює 3^2, то генерується виняток;
- на 4: x [: 2], x [2: 5], x [5: 8], x [8:];
- на 3 через підрядник: A [: 1,:], A [1: 3,:], A [3:,:];
- на 3 через підрядник: A [: 1,:], A [1: 3,:], A [3:,:];
- на 3 по стовпцях: A [:,: 1], A [:, 1: 3], A [:, 3:];
- через **підрядник**
- по **стовпцях**

```
x=np.arange(9)
np.split(x, 3)
Out[1]: [array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]

x2=np.arange(16)
np.split(x2, 4)
Out[2]:
        [array([0, 1, 2, 3]),
         array([4, 5, 6, 7]),
         array([ 8,  9, 10, 11]),
         array([12, 13, 14, 15])]

np.split(x, [2, 5, 8])
Out[3]: [array([0, 1]), array([2, 3, 4]), array([5, 6, 7]), array([8])]

A=np.arange(10).reshape((5,2))
A
Out[4]:
        array([[0, 1],
               [2, 3],
               [4, 5],
               [6, 7],
               [8, 9]])

np.split(A, [1, 3])
Out[5]:
```

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

np.split(A, [1, 3], axis=0)
Out[6]:
        [array([[0, 1]]),
         array([[2, 3], [4, 5]]),
        array([[6, 7], [8, 9]])]

np.split(A, [1, 3], axis=1)
Out[7]:
        [array([[0],
                [2],
                [4],
                [6],
                [8]]), array([[1],
                [3],
                [5],
                [7],
                [9]]), array([], shape=(5, 0), dtype=int32)]

np.vsplit(A, [1,3])
Out[53]:
        [array([[0, 1]]),
        array([[2, 3], [4, 5]]),
        array([[6, 7], [8, 9]])]

np.hsplit(A, [1,3])
Out[54]:
        [array([[0],
                [2],
                [4],
                [6],
                [8]]), array([[1],
                [3],
                [5],
                [7],
```

[9]]), array([], shape=(5, 0), dtype=int32)]

Функциія split працює трохи швидше функцій vsplit и hsplit.

Приклад

```
%timeit np.split(A, [1, 3], axis=0)
6.37 µs ± 28.2 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit np.vsplit(A, [1,3])
6.81 µs ± 14.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit np.split(A, [1, 3], axis=1)
6.35 µs ± 17.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

%timeit np.hsplit(A, [1,3])
6.96 µs ± 27.9 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

8. Копіювання масивів

a. Просте присвоювання

Присвоювання – це копіювання посилань.

Тут зміна значення елемента в масиві B **тягне** за собою зміну елемента в масиві A.

Приклад

```
A=np.array([[1,2],[3,4]])
B=A
B[0,0]=10
Out[1]:
      array([[10,  2],
             [ 3,  4]])
```

Це створення копії:

np.**copy**(a)

Тут: **a** – вхідний масив

Тут зміна значення елемента в масиві B **НЕ** впливає на масив A:

Приклад

```
B=np.copy(A)
B[0,0]=-5
B
Out[1]:
```

```
        array([[-5,  2],
               [ 3,  4]])

A
Out[2]:
        array([[10,  2],
               [ 3,  4]])
```

b. Лінеаризація

Лінеаризація – це теж копіювання посилань.

Повертається одновимірний масив, що містить елементи вводу. Копія робиться лише за потреби.

np.**ravel**(a)

Тут: **a** – вхідний масив.

Приклад

```
A=np.array([[0,1],[2,3]])
A
Out[1]:
        array([[0, 1],
               [2, 3]])

B=A.ravel()

B[1]=25
B
Out[2]: array([ 0, 25,  2,  3])

A
Out[3]:
        array([[ 0, 25],
               [ 2,  3]])
```

Метод ndarray.flatten () повертає копію масиву, стислу до одного виміру.

Тут також створюється копія:

a.**flatten**()

Тут: **a** – вхідний масив.

Приклад

```
B=A.flatten()

B[1]=500
B
Out[1]: array([  0, 500,   2,   3])
```

Ось тут зміна елементів в B **ніяк** не вплинуло на масив A:

```
A
Out[2]:
       array([[ 0, 25],
              [ 2,  3]])
```

c. Slicing – Нарізка (розшарування)

Нарізка в python означає перенесення елементів з одного заданого індексу до іншого заданого індексу. Ми передаємо фрагмент замість індексу таким чином: [початок: кінець]. Ми також можемо визначити крок таким чином: [start: end: step]. Якщо ми не пройдемо, почнемо його вважається 0. Якщо ми не пройдемо, закінчимо розглянуту довжину масиву в цьому вимірі. Якщо ми не пройдемо крок, його розглянемо як 1.

При slicing теж відбувається **копіювання посилань.**

Приклад

```
v=np.arange(5)
v
Out[1]: array([0, 1, 2, 3, 4])

v2=v[2:4]
v2
Out[2]: array([2, 3])
```

У цьому випадку зміна значення елемента в масиві v2 тягне за собою **зміну** елемента в масиві v:

```
v2[0]=22
v2
Out[3]: array([22,  3])

v
Out[4]: array([ 0,  1, 22,  3,  4])
```

Тут відбувається створення копії:
v2=v[2:4].copy()

У цьому випадку зміна значення елемента в масиві v2 **НЕ** впливає на масив v:
v2[0]=100
v2
Out[5]: array([100, 3])

v
Out[6]: array([0, 1, 22, 3, 4])

d. Вибірка по масці – створюється копія

Цей метод дозволяє встановити маску, з якою ми можемо вибрати відповідні їй елементи створивши новий масив.

При вибірці по масці створюється **копія.**

Приклад

v=np.arange(5)
v
Out[1]: array([0, 1, 2, 3, 4])

v3=v[v>2]
v3
Out[2]: array([3, 4])

v3[0]=33
v3
Out[3]: array([33, 4])

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

e. Fancy indexing – створюється копія

Цей метод дозволяє вибрати елементи масиву по їх індексами.

При fancy indexing створюється **копія**.

Приклад

v=np.arange(5)
v

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

v4=v[[1,3,0]]
v4
Out[2]: array([1, 3, 0])

v4[0]=11
v4
Out[3]: array([11,  3,  0])

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

f. Перетворення типів – створюється копія

ndarray.**astype**(dtype)

Тут: **dtype** – типовий код або тип даних, до якого відтворюється масив.

При перетворенні типу створюється копія:

Приклад

```
x=np.arange(5)
x
Out[1]: array([0, 1, 2, 3, 4])

y=x.astype(float)
y
Out[2]: array([ 0.,  1.,  2.,  3.,  4.])
```

У цьому випадку зміна значення елемента в масиві y **НЕ** впливає на масив x:

```
y[1]=100
y
Out[3]: array([   0.,  100.,    2.,    3.,    4.])
x
Out[4]: array([0, 1, 2, 3, 4])
```

8. Завдання та вправи

1. Обчислити суму матриці A і вектору x по стовбцям.

```
A=np.arange(20).reshape(5,4)
```

```
A
Out[25]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])

x=np.arange(5)
x
Out[27]: array([0, 1, 2, 3, 4])
```

2. За допомогою яких функцій можна отримати наступні матриці?

```
A
array([[ 5, -3, 10],
       [ 6,  8, -2]])
```

- array([[5, -3, 10],
  [5, -3, 10],
  [6, 8, -2],
  [6, 8, -2],
  [6, 8, -2],
  [6, 8, -2]])
- array([[5, 5, -3, -3, -3, 10, 10],
  [6, 6, 8, 8, 8, -2, -2]])
- array([[5, -3, 10, 5, -3, 10],
  [6, 8, -2, 6, 8, -2],
  [5, -3, 10, 5, -3, 10],
  [6, 8, -2, 6, 8, -2],
  [5, -3, 10, 5, -3, 10],
  [6, 8, -2, 6, 8, -2],
  [5, -3, 10, 5, -3, 10],
  [6, 8, -2, 6, 8, -2]])

3. Як за допомогою стандартних функцій numpy і функцій конкатенації можна заповнити матрицю, щоб вона прийняла даний вигляд?

```
array([[ 3.,  3.,  1.,  1.,  1., -0., -0., -3.],
       [ 3.,  3.,  0.,  1.,  1., -0., -3., -0.],
       [ 3.,  3.,  0.,  0.,  1., -3., -0., -0.],
       [ 2.,  6.,  0.,  1.,  6.,  4.,  2.,  0.],
       [ 0.,  2.,  6.,  2., -1., -1.,  8.,  0.],
```

[0., 0., 2., 3., -1., -1., 0., 8.]])

Рекомендована література

1. Robert Johansson. Introduction to Scientific Computing with Python, 2016.
2. Wes McKinney. Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython 2nd Edition, O'Reilly, 2018, 511 p.
3. Swaroop C. H. A byte of Python, 2013.
4. Travis E. Oliphant. Guide to NumPy, 2006.
5. https://www.python.org/
6. https://numpy.org/
7. https://matplotlib.org
8. https://pandas.pydata.org/docs/
9. https://www.sympy.org/en/index.html
10. https://scipy.org/about.html
11. https://python-scripts.com/pyqt5

Навчальне видання

Таірова Марія Сергіївна
Журавльова Зінаїда Юріївна

МОВА ПРОГРАМУВАННЯ PYTHON ДЛЯ НАУКОВИХ ОБЧИСЛЕНЬ
Частина 1

НАВЧАЛЬНИЙ ПОСІБНИК
з дисципліни «Програмні засоби наукових обчислень»
для студентів спеціальності 113 «прикладна математика»

В авторській редакції

Підп. до друку 14.03.2022. Формат 60x84/16.
Ум.-друк. арк. 14,85. Наклад 23 пр.
Зам. № 2497.

Видавець і виготовлювач
Одеський національний університет імені І. І. Мечникова
Свідоцтво суб'єкта видавничої справи ДК № 4215 від 22.11.2011 р.
65082, м. Одеса, вул. Єлісаветинська, 12, Україна
Тел.: (048) 723 28 39, e-mail: druk@onu.edu.ua