МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

С. Ю. Борю, Ю. М. Кревсун, Н. О. Миронова

ПРОГРАМУВАННЯ: ПОПУЛЯРНІ МОДУЛІ ТА ПАКЕТИ В PYTHON

Навчальний посібник для здобувачів ступеня вищої освіти
бакалавра спеціальності «Комп'ютерні науки»
освітньо-професійної програми «Комп'ютерні науки»

Затверджено
вченою радою ЗНУ
Протокол № 8 від 29.01.2026

Запоріжжя
2026

УДК 004 (076)
Б68

Борю С. Ю., Кревсун Ю. М., Миронова Н. О. Програмування: популярні модулі та пакети в Python : навчальний посібник для здобувачів ступеня вищої освіти бакалавра спеціальності «Комп’ютерні науки» освітньо-професійної програми «Комп’ютерні науки». Запоріжжя : Запорізький національний університет, 2026. 277 с.

Навчальний посібник «Програмування: популярні модулі та пакети в Python» спрямований на закріплення студентами теоретичних знань із дисципліни «Програмування» та формування навичок роботи з основними, часто використовуваними пакетами (модулями) системи програмування Python.

Видання охоплює ключові питання, передбачені силабусом дисципліни, і детально розглядає популярні модулі та пакети Python. Посібник містить необхідну теоретичну інформацію, велику кількість прикладів і фрагментів програм для практичного застосування набутих знань, а також контрольні запитання та тестові завдання для самоконтролю до кожної теми.

Навчальний посібник призначений для здобувачів ступеня вищої освіти бакалавра спеціальності «Комп'ютерні науки» освітньо-професійної програми «Комп’ютерні науки», але може бути корисним для студентів інших спеціальностей, які прагнуть систематизувати знання у сфері прикладного програмування та ефективного використання популярних бібліотек Python.

Рецензент
С. І. Гоменюк, доктор технічних наук, професор, декан математичного факультету

Відповідальний за випуск
О. С. Пшенична, кандидат педагогічних наук, в. о. завідувача кафедри комп’ютерних наук.

.

ВСТУП

Дисципліна «Програмування» належить до обов'язкових дисциплін циклу професійної підготовки спеціальності «Комп'ютерні науки» освітньо-професійної програми «Комп'ютерні науки».

Стандартна бібліотека Python містить сотні модулів, що дозволяють взаємодіяти з операційною системою, вирішувати найрізноманітніші завдання з обробки інформації. Усі вони ретельно протестовані і готові до негайного використання для розробки додатків. Переважна кількість пакетів поширюється безкоштовно.

Метою вивчення навчальної дисципліни «Програмування» є засвоєння студентами знань із теоретичних та практичних методів, алгоритмів та спеціальних підходів роботи з основних, часто використовуваних, пакетів (модулів) системи програмування Python. Опанування способів застосування, зазначених вище пакетів, в практиці проєктування і при реалізації сучасних програмних додатків і продуктів. Отримання навичок застосування обраних, які часто застосовуються в розробках, пакетів системи програмування Python.

Видання охоплює ключові питання. Детально розглядаються популярні модулі та пакети Python. Зокрема, висвітлюються такі бібліотеки та модулі: NumPy (для наукових обчислень та роботи з багатовимірними масивами), Matplotlib (для побудови графіків та візуалізації даних), SymPy (для символьних обчислень), tkinter (для створення графічного інтерфейсу додатків, GUI), стандартні модулі, які надають функції для математичних обчислень (math, fractions, cmath), роботи з бінарними та текстовими файлами (struct, csv, shelve), взаємодії з операційною системою (os, sys) та налагодження програм (datetime, logging, locale).

Основними **завданнями** вивчення дисципліни «Програмування» є засвоєння студентами теоретичних знань і набуття навичок застосування основних пакетів (модулів) системи програмування Python при проєктуванні й реалізації сучасних програмних продуктів.

Згідно з вимогами освітньої-професійної програми здобувачі освіти мають досягти таких компетентностей та програмних результатів навчання:

- здатність до абстрактного мислення, аналізу та синтезу;
- здатність застосовувати знання у практичних ситуаціях;
- знання та розуміння предметної області та розуміння професійної діяльності;
- здатність вчитися і оволодівати сучасними знаннями;
- здатність до логічного мислення, побудови логічних висновків, використання формальних мов і моделей алгоритмічних обчислень, проєктування, розроблення й аналізу алгоритмів, оцінювання їх ефективності та складності, розв'язності та нерозв'язності алгоритмічних проблем для адекватного моделювання предметних областей і створення програмних та інформаційних систем;
- здатність проєктувати та розробляти програмне забезпечення із застосуванням різних парадигм програмування: узагальненого, об'єктно-

орієнтованого, функціонального, логічного, з відповідними моделями, методами й алгоритмами обчислень, структурами даних і механізмами управління;

– застосовувати знання основних форм і законів абстрактно-логічного мислення, основ методології наукового пізнання, форм і методів вилучення, аналізу, обробки та синтезу інформації в предметній області комп'ютерних наук;

– використовувати сучасний математичний апарат неперервного та дискретного аналізу, лінійної алгебри, аналітичної геометрії, в професійній діяльності для розв’язання задач теоретичного та прикладного характеру в процесі проєктування та реалізації об’єктів інформатизації;

– проєктувати, розробляти та аналізувати алгоритми розв’язання обчислювальних та логічних задач, оцінювати ефективність та складність алгоритмів на основі застосування формальних моделей алгоритмів та обчислюваних функцій;

– розробляти програмні моделі предметних середовищ, вибирати парадигму програмування з позицій зручності та якості застосування для реалізації методів та алгоритмів розв’язання задач в галузі комп’ютерних наук.

Структура та зміст навчального посібника відповідають силабусу дисципліни «Програмування».

Навчальний посібник «Програмування: популярні модулі та пакети в Python» призначений для студентів спеціальності «Комп’ютерні науки» освітньо-професійної програми «Комп’ютерні науки», але представлене видання може бути корисним студентам різних спеціальностей.

1. МОДУЛІ ТА ПАКЕТИ В PYTHON

Встановлення python пакетів за допомогою pip

pip – це система керування пакетами, яка використовується для встановлення та керування програмними пакетами, написаними на Python. З версії Python 3.4, `pip` поставляється разом з інтерпретатором `python`.

Початок роботи

За допомогою `pip` встановимо якийсь пакет, наприклад, `numpy` (https://pythonworld.ru/numpy) у консолі (в режимі адміністратора):

```
Linux:
sudo pip install numpy

Windows:
pip install numpy
```

Може виникнути помилка: "`pip`" не є внутрішньою чи зовнішньою командою, яку виконує програма або пакетний файл.

Тоді необхідно вказувати повний шлях до програми, наприклад (якщо Python встановлений у C:\Python36):

```
C:\Python36\Tools\Scripts\pip3.exe install numpy,
```

або додавати папку C:\Python36\Tools\Scripts\ в PATH вручну, або зробивши цю папку з `pip3` поточною:

```
cd "C:\Python36\Tools\Scripts\"
pip install numpy
```

В останніх версіях Python виклик `pip` зручно виконувати з консолі:

```
python -m pip <command> [options]
```

Основні команди `pip`:

`pip help` – Допомога по доступним командам.

`pip install package_name` – Встановлення пакета(ів).

pip uninstall package_name – Видалення пакета(ів).

`pip list` – Перелік встановлених пакетів.

`pip show package_name` – Показує інформацію про встановлений пакет.

`pip search` – Пошук пакетів за ім'ям.

`pip -- proxy user: passwd@proxy.server :port` – Використання з проксі.

`pip install -U` – Оновлення пакета(ів).

`pip install --force-reinstall` – при оновленні, інсталювати пакет, навіть якщо він останньої версії.

Приклад перевірки, встановлення та переустановлення пакета `matplotlib`:

```
"C:\Program Files\Python36-32\Scripts\pip" install -U matplotlib
```

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC
v.1900 32 bit (Intel)] on win32
Тип "copyright", "credits" або "license()" for more information.
>>> import matplotlib as mpl
>>> print ('Current version on matplotlib library is',
mpl.__version__)

Current version on matplotlib library is 2.1.1
>>>
# де-факто стандарт виклику pyplot у python
import matplotlib.pyplot as plt
```

Модулі в Python

Система модулів дозволяє логічне організувати код Python. Групування коду в модулі значно полегшує процес написання та розуміння програми. Модуль Python це просто файл, що містить код Python. Кожен модуль Python може містити змінні, оголошення класів і функцій. Крім того, у модулі може знаходитися виконуваний код.

Кожна програма може імпортувати модуль та отримати доступ до його класів, функцій та об'єктів. Потрібно помітити, що модуль може бути написаний не лише на Python, а, наприклад, на C або C++.

Підключення модуля зі стандартної бібліотеки

Підключити модуль можна за допомогою інструкції import. Наприклад, підключимо модуль os для отримання поточної директорії:

```
>>> import os
>>> os.getcwd()
'C:\\Python33'
```

Після ключового слова `import` вказується назва модуля. Однією інструкцією можна підключити кілька модулів, хоча цього не рекомендується робити, оскільки це знижує читання коду. Імпортуємо модулі time та random.

```
>>>
>>> import time,random
>>> time.time()
1376047104.056417
>>> random.random()
0.9874550833306869
```

Після імпортування модуля його назва стає змінною, через яку можна отримати доступ до атрибутів модуля. Наприклад, можна звернутися до константи e, розташованої в модулі `math`:

```
>>>
>>> import math
>>> math.e
2.718281828459045
```

Варто зазначити, що якщо зазначений атрибут модуля не буде знайдено, збудіться виняток `AttributeError`. Якщо ж не вдасться знайти модуль для імпортування, то `ImportError`.

```
>>>
```

```
>>> import notexist
Traceback (most recent call last):
 File "", line 1, in
import notexist
importError: No module named 'notexist'
>>> import math
>>> math.
Traceback (most recent call last):
 File "", line 1, in
 math.
AttributeError: 'module' object has no attribute 'E'
```

Використання псевдонімів

Якщо назва модуля занадто довга, або вона вам не подобається з якихось інших причин, то для нього можна створити псевдонім за допомогою ключового слова as.

```
>>>
>>> import math as m
>>> m.e
2.718281828459045
```

Тепер доступ до всіх атрибутів модуля `math` здійснюється лише за допомогою змінної `m`, а змінної `math` у цій програмі вже не буде (якщо, звичайно, ви після цього не напишете `import math`, тоді модуль буде доступний як під ім'ям m, так і під ім'ям `math`).

Інструкція from

Підключити певні атрибути модуля можна за допомогою інструкції від. Вона має кілька форматів:

```
from <Назва модуля> import <Атрибут 1>
                          [as <Псевдонім 1> ],
                     [<Атрибут 2>
                        [as <Псевдонім 2> ] ...]

from <Назва модуля> import *
```

Перший формат дозволяє підключити з модуля лише вказані атрибути. Для довгих імен можна призначити псевдонім, вказавши його після ключового слова as.

```
>>>
>>> from math import e, ceil as c
>>> e
2.718281828459045
>>> c(4.6)
5
```

Атрибути, що імпортуються, можна розмістити на декількох рядках, якщо їх багато (для кращої читання коду):

```
>>>
>>> from math import(sin, cos,
..                 . tan, atan)
```

Другий формат інструкції `from` дозволяє підключити всі (точніше, майже всі) змінні з модуля. Наприклад, імпортуємо всі атрибути з модуля `sys`:

```
>>> from sys import *
>>> version
'3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.1600 32
bit (Intel)]'
>>> version_info
sys.version_info (major=3, minor=3, micro=2, releaselevel='final',
serial=0)
```

Слід зазначити, що не всі атрибути імпортуватимуться. Якщо в модулі визначено змінну `__all__` (список атрибутів, які можуть бути підключені), то будуть підключені лише атрибути цього списку. Якщо змінна `__all__` не визначена, то буде підключено всі атрибути, що не починаються з нижнього підкреслення.

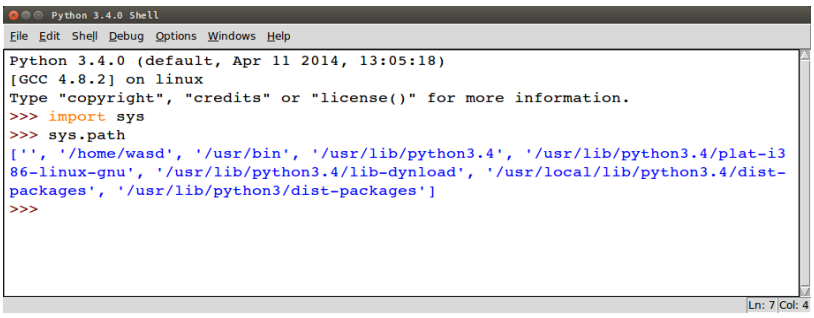
Крім того, необхідно враховувати, що імпортування всіх атрибутів з модуля може порушити простір імен головної програми, оскільки змінні, що мають однакові імена, будуть перезаписані.

Місце знаходження модулів у Python:

Коли імпортується модуль, інтерпретатор Python шукає цей модуль у таких місцях:

1. Директорія, в якій знаходиться файл, у якому викликається команда імпорту
2. Якщо модуль не знайдено, Python шукає у кожній директорії, визначеній у консольній змінній `PYTHONPATH`.
3. Якщо і там модуль не знайдено, Python перевіряє шлях, заданий за замовчуванням.

Шлях пошуку модулів збережено в системному модулі `sys` змінної `path`. Змінна `sys.path` містить усі три вищеописані місця пошуку модулів:

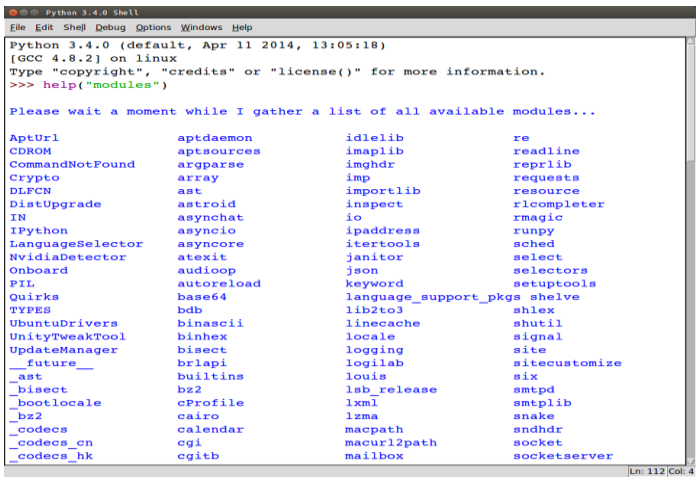


Отримання списку всіх модулів Python, встановлених на комп'ютері

Для того, щоб отримати список усіх модулів, встановлених на комп'ютері, достатньо виконати команду:

```
help("modules")
```

За кілька секунд виведеться список усіх доступних модулів:



Створення модуля в Python:

Щоб створити свій модуль у Python, достатньо зберегти скрипіт з розширенням `.py` Тепер він доступний в будь-якому іншому файлі. Наприклад, створимо два файли: `module_1.py` та `module_2.py` і збережемо їх в одній директорії. У першому напишемо:

```
def hello():
    print("Hello from module_1")
```

А в другому викличемо цю функцію:

```
from module_1 import hello

hello()
```

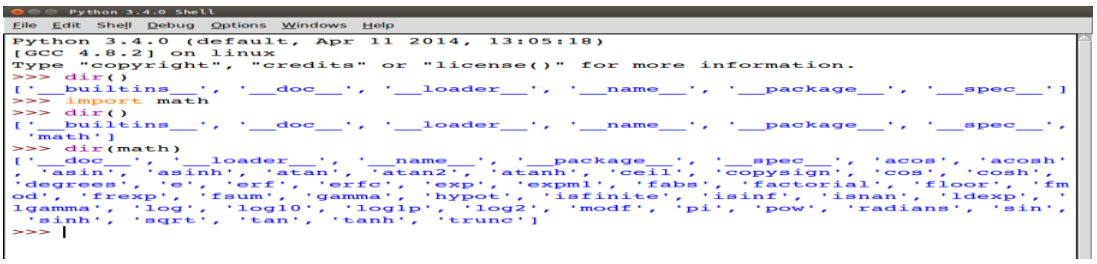
Виконавши код другого файлу отримаємо:

```
Hello from module_1
```

Функція dir()

Вбудована функція `dir()` повертає відсортований список рядків, що містять усі імена, визначені у модулі:

```
# на даний момент нам доступні лише вбудовані функції
dir()
# Імпортуємо модуль math
import math
# тепер модуль math у списку доступних імен
dir()
# отримаємо імена, визначені в модулі math
dir(math)
```



Архітектура програми на Python

Код Python може бути організований таким чином:

1. Перший рівень – це звичайні команди на Python.
2. Команди на Python можуть бути зібрані у функції.
3. Функції може бути частиною класу.
4. Класи можна визначити всередині модулів.
5. Нарешті, модулі можуть складатися пакети модулів.

Пакети модулів у Python

Окремі файли-модулі з кодом Python можуть об'єднуватися в пакети модулів. Пакет це директорія (папка), що містить кілька окремих файлів-скриптів.

Наприклад, маємо таку структуру:

```
|__ my_file.py
|__ my_package
      |_ __init__.py
      |_ inside_file.py
```

У файлі `inside_file.py` визначено певну функцію `foo`. Тоді, щоб отримати доступ до функції `foo`, у файлі `my_file` слід виконати наступний код:

```
from my_package.inside_file import foo
```

Також слід звернути увагу на наявність усередині директорії `my_package` файлу `__init__.py`. Це може бути порожній файл, який повідомляє Python, що ця директорія є пакетом модулів. У Python 3.3 і вище включати файл `__init__.py` в пакет модулів стало необов'язково, проте рекомендується робити це задля підтримки зворотної сумісності.

Питання для самоконтролю до теми 1

1. Для чого потрібна програма `pip`?
2. Як підключити модуль із стандартної бібліотеки
3. Що таке псевдонім модуля і як його можна використовувати?
4. Як отримати список усіх модулів, встановлених у системі програмування Python на робочій станції?
5. Як створити модуль користувача?

Завдання до теми 1

Створіть на робочому столі папку `project` з двома файлами: `main.py` та `calculator.py`. У файлі `calculator.py` визначте просту функцію `add(a, b)`, що повертає суму двох чисел. Використайте консольну команду `pip` для встановлення будь-якого зовнішнього пакету (наприклад, `requests`). Після цього у `main.py` імпортуйте функцію `add` з `calculator.py` і викличте її. Також імпортуйте встановлений пакет `requests`, використовуючи псевдонім `r`, і виведіть його версію, щоб підтвердити успішне імпортування обох модулів.

2. БІБЛІОТЕКА NUMPY

NumPy початок роботи

NumPy – це бібліотека мови Python, яка додає підтримку великих багатовимірних масивів і матриць, разом із великою бібліотекою високорівневих (і дуже швидких) математичних функцій для операцій із цими масивами.

Встановлення NumPy

Linux:

```
sudo pip install numpy
```

Windows:

```
pip install numpy
```

Найважливіші атрибути

Основним об'єктом NumPy є однорідний багатовимірний масив (numpy називається numpy.ndarray). Це багатовимірний масив елементів (зазвичай чисел), одного типу.

Найбільш важливі атрибути об'єктів ndarray:

ndarray.ndim – Число вимірювань (частіше їх називають "осі") масиву.

ndarray.shape – Розміри масиву, його форма. Це кортеж натуральних чисел, що показує довжину масиву кожної осі. Для матриці з n рядків та m стовпів, shape буде (n,m). Число елементів кортежу shape дорівнює ndim.

ndarray.size – Кількість елементів масиву. Очевидно, дорівнює добутку всіх елементів атрибуту shape.

ndarray.dtype – Об'єкт, що описує тип елементів масиву. Можна визначити dtype за допомогою стандартних типів даних Python. NumPy тут надає цілий букет можливостей, як вбудованих, наприклад: bool, character, int8, int16, int32, int64, float8, float16, float32, float64, complex64, object, і можливість визначити власні типи даних, зокрема і складові.

ndarray.itemsize – Розмір кожного елемента масиву в байтах.

ndarray.data – Буфер, що містить фактичні елементи масиву. Зазвичай не потрібно використовувати цей атрибут, тому що звертатися до елементів масиву найпростіше за допомогою індексів.

Створення масивів

У NumPy є багато способів створити масив.

Один з найпростіших – створити масив зі звичайних списків або кортежів Python. Для цього використовується функція numpy.array() (array – функція, що створює об'єкт типу ndarray):

```
>>>
```

```
>>> import numpy as np
>>> a =np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> type(a)
<class 'numpy.ndarray'>
```

Функція `array`() трансформує вкладені послідовності багатомірні масиви. Тип елементів масиву залежить від типу елементів вихідної послідовності (але можна і перевизначити його на момент створення).

```
>>>
>>> b = np.array([[1.5, 2, 3], [4, 5, 6]])
>>> b
array([[ 1.5, 2. , 3. ],
 [4., 5., 6.]])
```

Можна також перевизначити тип у момент створення:

```
>>>
>>> b = np.array([[1.5, 2, 3], [4, 5, 6]], dtype=np.complex)
>>> b
array([[ 1.5+0.j, 2.0+0.j, 3.0+0.j],
 [ 4.0+0.j, 5.0+0.j, 6.0+0.j]])
```

Функція `array`() не єдина функція створення масивів. Зазвичай елементи масиву спочатку невідомі, а масив, у якому зберігатимуться, вже потрібний. Тому є кілька функцій для того, щоб створювати масиви з якимось вихідним вмістом (за замовчуванням тип масиву, що створюється – `float64`).

Функція `zeros()` – створює масив з нулів, а функція `ones()` – створює масив з одиниць.

Обидві функції приймають кортеж з розмірами та аргумент `dtype`:

```
>>>
>>> np.zeros((3, 5))
array([[ 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0.],
 [ 0., 0., 0., 0., 0.]])
>>> np.ones((2, 2, 2))
array([[[ 1., 1.],
 [1., 1.]],

 [[ 1., 1.],
 [1., 1.]]])
```

Функція `eye()` створює одиничну матрицю (двовимірний масив)

```
>>>
>>> np.eye(5)
array([[ 1., 0., 0., 0., 0.],
 [0., 1., 0., 0., 0.],
 [0., 0., 1., 0., 0.],
 [0., 0., 0., 1., 0.],
 [ 0., 0., 0., 0., 1.]])
```

Функція `empty()` створює масив без заповнення. Вихідний вміст випадково і залежить від стану пам'яті на момент створення масиву (тобто від сміття, що в ній зберігається):

```
>>>
>>> np.empty((3, 3))
array([[ 6.93920488e-310, 6.93920488e-310, 6.93920149e-310],
 [6.93920058e-310, 6.93920058e-310, 6.93920058e-310],
 [6.93920359e-310, 0.00000000e+000, 6.93920501e-310]])
>>> np.empty((3, 3))
array([[ 6.93920488e-310, 6.93920488e-310, 6.93920147e-310],
 [6.93920149e-310, 6.93920146e-310, 6.93920359e-310],
 [6.93920359e-310, 0.00000000e+000, 3.95252517e-322]])
```

Повний формат виклику функцій:

```
numpy.zeros(shape, dtype = float, order = 'C')
numpy.ones(shape, dtype = float, order = 'C')
numpy.empty(shape, dtype = float, order = 'C')
```

де:

`shape` – Обов'язковий аргумент, кортеж необхідної розмірності масиву;

`dtype` – опціональний аргумент, тип елементів масиву, за замовчанням `float`;

`order` – опціональний аргумент, рядок, спосіб представлення даних масиву в пам'яті, два можливі значення 'C' і 'F' – "як у C" або "як у фортрані".

Для створення послідовностей чисел, NumPy є функція

`arange()`**,** аналогічна до вбудованої в Python `range`(), тільки замість списків вона повертає масиви, і приймає не тільки цілі значення:

```
>>>
>>> np.arange(10, 30, 5)
array([10, 15, 20, 25])
>>> np.arange(0, 1, 0.1)
array([0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

Повний формат виклику функції:

```
numpy.arange([start,] stop[, step,], dtype = None),
```

де:

`start` – опціональний аргумент, початкове значення інтервалу, за умовчанням дорівнює 0;

`stop` – обов'язковий аргумент, кінцеве значення інтервалу, що не входить до самого інтервалу, інтервал замикає значення `stop` - `step`;

`step` – опціональний аргумент, крок ітерації, різницю між кожним наступним та попереднім значеннями інтервалу, за умовчанням дорівнює 1;

`dtype` – тип елементів масиву, за замовчуванням `None`, у разі тип елементів визначається за типом переданих функції аргументів `start`, `stop`.

Масив може бути створений за допомогою функції `numpy.linspace()`.

Взагалі, при використанні `arange()` з аргументами типу `float`, складно бути впевненим у тому, скільки елементів буде отримано (через обмеження

точності чисел із рухомою комою). Тому в таких випадках зазвичай краще використовувати функцію:

`linspace()`, *яка замість кроку в якості одного з аргументів приймає число, що дорівнює кількості потрібних елементів*:

```
>>>
>>> np.linspace(0, 2, 9) #9 чисел від 0 до 2 включно
array([0., 0.25, 0.5, 0.75, 1., 1.25, 1.5, 1.75, 2.])
```

Повний формат виклику функції:

`numpy.linspace`(start, stop, num = 50, endpoint = True, retstep = False),

де:

`start` – Обов'язковий аргумент, перший член послідовності елементів масиву;

`stop` – Обов'язковий аргумент, останній член послідовності елементів масиву;

`num` – опціональний аргумент, кількість елементів масиву, за умовчанням дорівнює 50;

`endpoint` – опціональний аргумент, логічне значення за умовчанням True. Якщо передано True, то stop є останнім елементом масиву. Якщо встановлено False, то послідовність елементів формується від start до stop для num + 1 елементів, при цьому повертається масив останній елемент не входить;

```
>>> d = np.linspace (1.0, 6.0, 5, endpoint = False)
>>> d
array([ 1., 2., 3., 4., 5.])
```

`retstep` – опціональний аргумент, логічне значення за замовчуванням `False`. Якщо передано `True` то, функція повертає кортеж із двох членів, перший - масив, послідовність елементів, другий – число, збільшення між елементами послідовності.

```
>>> f = np.linspace(.5, -.5, 5, retstep = True)
>>> f
(array([ 0.5 , 0.25, 0. , -0.25, -0.5 ]), -0.25)
```

```
fromfunction() – застосовує функцію до всіх комбінацій індексів
>>>
>>> def f1(i, j):
... return 3 * i + j
...
>>> np.fromfunction(f1, (3, 4))
array([[ 0., 1., 2., 3.],
 [3., 4., 5., 6.],
 [6., 7., 8., 9.]])

>>> np.fromfunction(f1, (3, 3))
array([[ 0., 1., 2.],
 [3., 4., 5.],
 [6., 7., 8.]])
```

Ну і останній з цих способів – за допомогою функцій `numpy.zeros_like()`, `numpy.ones_like()`, `numpy.empty_like()`.

Обов'язковий аргумент, що приймається функціями – масив, функції повертають масив такої ж структури, що містить, відповідно, нулі, одиниці та те, що виявилося в пам'яті на момент створення масиву.

```
>>> d=array([ 1., 2., 3., 4., 5.])

>>> dz = np.zeros_like(d)
>>> dz
array([0., 0., 0., 0., 0.])

>>> do = np.ones_like(d)
>>> do
array([ 1., 1., 1., 1., 1.])

>>> de = np.empty_like(d)
>>> de
array([ 2.24122267e+201, 6.32526950e-317,
 0.00000000e+000,
 2.24686637e+201, 6.34874355e-321])
```

Повний формат виклику функцій:

```
numpy.zeros_like(a)
numpy.ones_like(a)
numpy.empty_like(a)
```

де:

`a` – обов'язковий аргумент, масив, структуру якого необхідно повторити;

Масиви можна використовувати у різних ітераціях:

```
>>> for i in a:
... print(i)
...
0.1
0.2
0.3
0.4
0.5
```

```
>>> for i in a3:
..  . for j in i:
..    . print(j)
...
[1 2]
[3 4]
[5 6]
[7 8]
[9, 10]
[11 12]
```

Друк масивів

Якщо масив занадто великий, щоб друкувати, NumPy автоматично приховує центральну частину масиву і виводить тільки його куточки.

```
>>>
>>> print(np.arange(0, 3000, 1))
[ 0 1 2 ..., 2997 2998 2999]
```

Якщо потрібно побачити весь масив, використовуйте функцію `numpy.set_printoptions`:

```
np.set_printoptions(threshold=np.nan)
```

За допомогою цієї функції можна налаштувати друк масивів «під себе».

Функція `numpy.set_`printoptions приймає кілька аргументів:

`precision` – кількість цифр, що відображаються після коми (за замовчуванням 8).

`threshold` – кількість елементів масиву, що викликає обрізання елементів (за замовчуванням 1000).

`edgeitems` – кількість елементів на початку та наприкінці кожної розмірності масиву (за замовчуванням 3).

`linewidth` – кількість символів у рядку, після якого здійснюється перенесення (за умовчанням 75).

`suppress` – якщо `True`, не друкує невеликі значення в `non-scientific notation` (за замовчуванням `False`).

`nanstr` – строкове представлення `NaN` (за умовчанням `'nan'`)

`infstr` – строкове представлення `inf` (за умовчанням `'inf'`)

`formatter` – дозволяє більш тонко керувати печаткою масивів. Тут не розглядаємо – дивіться інформаційні ресурси [1, 2].

Базові операції над масивами

Математичні операції над масивами виконуються поелементно. Створюється новий масив, що заповнюється результатами дії оператора.

```
>>>
>>> import numpy as np
>>> a =np.array([20, 30, 40, 50])
>>> b =np.arange(4)
>>> a+b
array([20, 31, 42, 53])
>>> a-b
array([20, 29, 38, 47])
>>> a*b
array([0, 30, 80, 150])
>>> a/b # При розподілі на 0 повертається inf (нескінченність)
array([ inf, 30. , 20. , 16.66666667])
<string>:1: RuntimeWarning: divide by zero об'єднаний в true_divide
>>> a**b
array([1, 30, 1600, 125000])
>>> a%b # При взятті залишку від поділу на 0 повертається 0
<string>:1: RuntimeWarning: divide by zero об'єднаний в remainder
array([0, 0, 0, 2])
```

Для цього, природно, масиви мають бути однакових розмірів.

```
>>>
>>> c =np.array([[1, 2, 3], [4, 5, 6]])
```

```
>>> d =np.array([[1, 2], [3, 4], [5, 6]])
>>> c+d
Traceback (most recent call last):
 File "<input>", line 1, in <module>
ValueError: Operands could not be broadcast together with shapes
(2,3) (3,2)
```

Також можна проводити математичні операції між масивом та числом. У цьому випадку до кожного елемента додається (віднімається тощо) це число.

```
>>>
>>> a + 1
array([21, 31, 41, 51])
>>> a ** 3
array([8000, 27000, 64000, 125000])
>>> a < 35 # І фільтрацію можна проводити
array([True, True, False, False], dtype=bool)
```

NumPy також надає безліч математичних операцій для обробки масивів:

```
>>>
>>> np.cos(a)
array([0.40808206, 0.15425145, -0.66693806, 0.96496603])
>>> np.arctan(a)
array([1.52083793, 1.53747533, 1.54580153, 1.55079899])
>>> np.sinh(a)
array([ 2.42582598e+08, 5.34323729e+12, 1.17692633e+17,
 2.59235276e+21])
```

Повний список базових операцій можна переглянути в [2].

Список корисних математичних функцій пакета NumPy

Усі наступні функції, включно з `numpy.abs(a)`, можуть приймати як вхідні дані як одне число (скаляр), так і цілий масив чисел.

`numpy.abs(a)` – абсолютне значення a;

`numpy.around(a, decimals = 0, out = None)` – округлює a до заданої кількості десяткових розрядів, за замовчуванням до цілого. Дотримується наступного правила заокруглення. Якщо значення a знаходиться точно посередині між двома цілими, округлення проводиться до найближчого парного цілого. Так, якщо a дорівнює 1.5 або 2.5 буде повернуто 2, якщо a дорівнює -0.5 або 0.5 буде повернуто 0.0. Аргумент `decimals` – ціле, десятковий розряд після коми, до якого проводиться округлення, якщо decimals негативне, розряд відраховується ліворуч від коми.

```
print(np.around)(np.array([0.5, 1.8, 2.5, 3.5])))
[ 0. 2. 2. 4.]

print(np.around(np.array ([1, 5, 15, 45]), decimals = 1))
[ 1 5 15 45]
```

```
print(np.around)(np.array ([1, 5, 15, 45]), decimals=-1))
[ 0 0 20 40]
```

Аргумент out – масив, в який буде передано результат, структура масиву out, повинна збігатися зі структурою масиву, що повертається. Якщо None (за замовчуванням) буде створено новий масив.

`numpy.fix(a, out = None)` – відкидає дробову частину a;

`numpy.ceil(a, out = None)` – округлює a до меншого з цілих великих або рівних a;

```
>>> print(np.ceil(np.array([-2.7, -1.2, -0.5, 1.8, 2.4,
                                                3.6])))))
[-2. -1. -0. 2. 3. 4.]
```

`numpy.floor(a, out = None)` – округлює a до більшого з цілих менших або рівних a;

```
>>> print np.floor(np.array([-2.7, -1.2, -0.5, 1.8, 2.4,
                                                 3.6])))
[-3. -2. -1. 1. 2. 3.]
```

`numpy.sign(a)` – повертає -1 якщо a < 0, 0 якщо a == 0, 1 якщо a > 0;

`numpy.degrees(a)` – конвертує a з радіан у градуси;

`numpy.radians(a)` – конвертує a із градусів у радіани;

`numpy.cos(a), numpy.sin(a), numpy.tan(a)` – повертає косинус, синус, тангенс a (a у радіанах);

`numpy.cosh(a), numpy.sinh(a), numpy.tanh(a)` – повертає гіперболічний косинус, синус, тангенс a (a у радіанах);

`numpy.arccos(a), numpy.arcsin(a), numpy.arctan(a)` – повертає арккосинус, арксинус, арктангенс a в радіанах. Для арккосинусу в діапазоні [0, `nump.pi`], для арксинусу `[-numpy.pi/2, numpy.pi/2]`, для арктангенса [-`numpy.pi/2`, `numpy.pi/2`];

`numpy.arccosh(a), numpy.arcsinh(a), numpy.arctanh(a)` – повертає гіперболічний косинус, синус, тангенс a. a у радіанах;

`numpy.exp(a)` – повертає основу натурального логарифму (число e) у ступені a;

`numpy.log(a), numpy.log10(a), numpy.log2(a)` – повертає натуральний логарифм a, логарифм a на підставі 10, логарифм a на підставі 2;

`numpy.log1p(a)` – повертає натуральний логарифм `a + 1`;

`numpy.sqrt(a)` – повертає позитивний квадратний корінь a;

`numpy.square(a)` – повертає квадрат a.

У вирази можна включати кілька масивів. Якщо атрибути `ndarray.shape` цих масивів збігаються, результат очевидний – дії над масивами будуть проводитися поелементно:

```
>>> a1 = np.array([ [1.0, 2.0], [3.0, 4.0] ])
>>> a2 = np.array([ [5.0, 6.0], [7.0, 8.0] ])
>>> a3 = np.array([ [9.0, 10.0], [11.0, 12.0] ])
```

```
>>> print (a1 + a2 + a3)
[[ 15. 18.]
 [21. 24.]]

>>> print (a1 * a2)
[[ 5. 12.]
 [21. 32.]]

>>> print a3 / a1
[[ 9. 5. ]
 [ 3.66666667 3. ]]

>>> print (a3 - a1) * a2
[[ 40. 48.]
 [56. 64.]]
```

Якщо атрибути `ndarray.shape` не збігаються – дії над масивами виконуються відповідно до концепції транслювання (`broadcasting`).

Операція транслювання – розширення одного або обох масивів операндів до масивів з рівною розмірністю.

Для початку кілька прикладів:

```
>>> a2 = np.array([1, 2])
>>> print a2
[1 2]
>>> print a2.shape
(2,)
>>> a22 = np.array([ [1, 2], [3, 4] ])
>>> print a22
[[1 2]
 [3 4]]
>>> print a22.shape
(2, 2)

>>> a32 = np.array([ [1, 2], [3, 4], [5, 6] ])
>>> print a32
[[1 2]
 [3 4]
 [5 6]]
>>> print a32.shape
(3, 2)

>>> a222 = np.array([ [ [1, 2], [3, 4] ], [ [5, 6], [7, 8] ] ])
>>> print a222
[[[1 2]
 [3 4]]

 [[5 6]
 [7 8]]]
>>> print a222.shape
(2, 2, 2)

>>> print a22 + a2
[[2 4]
```

```
 [4 6]]

>>> print a32 + a2
[[2 4]
 [4 6]
 [6 8]]

>>> print a222 + a2
[[[ 2 4]
 [4 6]]

 [[ 6 8]
 [8, 10]]]

>>> print a32 + a22
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a
single shape

>>> print a222 + a22
[[[ 2 4]
 [6, 8]]

 [[ 6 8]
 [10 12]]]

>>> print a222 + a32
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a
single shape
```

Якщо, порівнюючи розміри двох масивів справа наліво, не знайдеться нерівних чисел, які б не дорівнювали одиниці, то транслювання відбудеться успішно. В іншому випадку система видасть помилку.

```
>>> a = np.ones((7, 3, 4, 9, 8))
>>> b = np.ones((4, 9, 8))
>>> c = np.ones((4, 3, 8))

>>> print (a + b).shape
(7, 3, 4, 9, 8)

>>> print (a + c).shape
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a
single shape

>>> d = np.ones((9, 7, 1, 6, 1))
>>> e = np.ones((7, 2, 1, 4))
```

```
>>> print (d + e).shape
(9, 7, 2, 6, 4)
```

Якщо в масивах присутні осі одиничної довжини, можуть розширюватися обидва масиви.

```
>>> f = np.array([1, 2])
>>> print f
[1 2]
>>> print f.shape
(2,)

>>> g = np.array ([[3], [4], [5]])
>>> print g
[[3]
 [4]
 [5]]
>>> print g.shape
(3, 1)

>>> h = f + g
>>> print h
[[4 5]
 [5 6]
 [6 7]]
>>> print h.shape
(3, 2)
```

Трансляція масивів відбувається при виклику деяких функцій.

Наприклад, функція `numpy.power(a1, a2)`, де `a1`, `a2` масив або скаляр, повертає `a1` до ступеня `a2`:

```
>>> print np.power(np.array([-2.0, -1.0, 0.0, 2.0, 3.0,\
                                                4.0]), 4)
[16. 1. 0. 16. 81. 256.]

>>> print np.power(np.array([-2.0, -1.0, 0.0, 2.0, 3.0,\
                                               4.0]), -4)
[0.0625 1. Inf 0.0625 0.01234568 0.00390625]
```

якщо у переданих масивів не збігається структура, проводить трансляцію.

```
>>>print np.power(np.array([3, 7]), np.array([ [1], [2],\
                                                [3] ])))
[[ 3 7]
 [949]
 [27343]]
```

Багато унарних операцій, такі як, наприклад, обчислення суми всіх елементів масиву, представлені також і у вигляді методів класу ndarray.

```
>>>
>>> a =np.array([[1, 2, 3], [4, 5, 6]])
>>> np.sum(a)
21
>>> a.sum()
21
```

```
>>> a.min()
1
>>> a.max()
6
```

За замовчуванням, ці операції застосовуються до масиву, ніби він був списком чисел, незалежно від його форми. Однак, вказавши параметр axis, можна застосувати операцію зазначеної осі масиву:

```
>>>
>>> a.min(axis=0)
 # Найменша кількість у кожному стовпці
array([1, 2, 3])
>>> a.min(axis=1)
 # Найменша кількість у кожному рядку
array([1, 4])
```

Індекси, зрізи, ітерації

Одномірні масиви здійснюють операції індексування, зрізів та ітерацій дуже схожим чином із звичайними списками та іншими послідовностями Python (хіба що видаляти за допомогою зрізів не можна).

```
>>>
>>> a =np.arange(10) ** 3
>>> a
array([0, 1, 8, 27, 64, 125, 216, 343, 512, 729])
>>> a[1]
1
>>> a[3:7]
array([ 27, 64, 125,216])
>>> a[3:7] = 8
>>> a
array([ 0, 1, 8, 8, 8, 8, 8, 343, 512, 729])
>>> a[::-1]
array([729, 512, 343, 8, 8, 8, 8, 8, 1, 0])
>>> del a[4:6]
Traceback (most recent call last):
 File "<input>", line 1, in <module>
ValueError: cannot delete array elements
>>> for i in a:
...print(i ** (1/3))
0.0
1.0
2.0
2.0
2.0
2.0
2.0
7.0
8.0
9.0
```

Багатомірні масиви на кожну вісь мають один індекс. Індекси передаються у вигляді послідовності чисел, розділених комами (тобто кортежами):

```
>>>
```

```
>>> b =np.array([[ 0, 1, 2, 3],
... [10, 11, 12, 13],
... [20, 21, 22, 23],
... [30, 31, 32, 33],
... [40, 41, 42, 43]])
...
>>> b[2,3] # Другий рядок, третій стовпець
23
>>> b[(2,3)]
23
>>> b[2][3] # Можна і так
23
>>> b[:,2] # Третій стовпець
array([2, 12, 22, 32, 42])
>>> b[:2] # Перші два рядки
array([[ 0, 1, 2, 3],
 [10, 11, 12, 13]])
>>> b[1:3, : : ] # Другий і третій рядки
array([[10, 11, 12, 13],
 [20, 21, 22, 23]])
```

Коли індексів менше, ніж осей, відсутні індекси передбачаються доповненими за допомогою зрізів:

```
>>>
>>> b[-1] # Останній рядок. Еквівалентно b[-1,:]

array([40, 41, 42, 43])
```

`b[i]` можна читати як

`b[i, <стільки символів ':', скільки потрібно>]`.

У NumPy це може бути записано з допомогою точок, як `b[i, ...]`.

Наприклад, якщо x має ранг 5 (тобто має 5 осей), тоді

`x[1, 2, ...]` еквівалентно `x[1, 2, :, :, :]`,

`x[... , 3]` те саме, що `x[:, :, :, :, 3] i`

`x[4, ... , 5, :]` це `x[4, :, :, 5, :]`.

```
>>>
>>> a =np.array(([[0, 1, 2], [10, 12, 13]], [[100, 101, 102],
[110, 112, 113]]))
>>> a.shape
(2, 2, 3)
>>> a[1, ...] # те саме, що a[1, : , :] або a[1]

array([[100, 101, 102],
 [110, 112, 113]])

>>> c[... ,2] # те, що a[: , : ,2]
array([[ 2, 13],
 [102, 113]])
```

Ітерування багатовимірних масивів починається з першої осі:

```
>>>
>>> for row in a:
..   .print(row)
...
[[ 0 1 2]
 [10 12 13]]
[[100 101 102]
 [110112113]]
```

Однак, якщо потрібно перебрати поелементно весь масив, ніби він був одномірним, для цього можна використовувати атрибут `flat`:

```
>>>
>>> for el in a.flat:
..   .print(el)
...
0
1
2
10
12
13
100
101
102
110
112
113
```

Маніпуляції із формою

Масив має форму (`shape`), яка визначається числом елементів уздовж кожної осі:

```
>>>
>>> a
array([[[ 0, 1, 2],
 [10, 12, 13]],

 [[100, 101, 102],
 [110, 112, 113]]])
>>> a.shape
(2, 2, 3)
```

Форма масиву може бути змінена за допомогою різних команд:

```
>>>
>>> a.ravel() # Робить масив плоским
array([ 0, 1, 2, 10, 12, 13, 100, 101, 102, 110, 112, 113])

>>> a.shape = (6, 2) # Зміна форми
>>> a
array([[ 0, 1],
 [2, 10],
 [12, 13],
 [100, 101],
```

```
 [102, 110],
 [112, 113]])

>>> a.transpose() # Транспонування
array([[ 0, 2, 12, 100, 102, 112],
 [1, 10, 13, 101, 110, 113]])
>>> a.reshape((3, 4)) # Зміна форми
array([[ 0, 1, 2, 10],
 [12, 13, 100, 101],
 [102, 110, 112, 113]])
```

Порядок елементів в масиві в результаті функції `ravel()` відповідає звичайному «С-стилю», тобто чим правіше індекс, тим він «швидше змінюється»: за елементом `a[0,0]` слідує `a[0,1]`.

Якщо одна форма масиву була змінена на іншу, масив також переформовується в «С-стилі».

Функції `ravel()` і `reshape()` також можуть працювати (при використанні додаткового аргументу) у FORTRAN-стилі, в якому швидше змінюється лівіший індекс.

```
>>>
>>> a
array([[ 0, 1],
 [2, 10],
 [12, 13],
 [100, 101],
 [102, 110],
 [112, 113]])
>>> a.reshape((3, 4),order='F')
array([[ 0, 100, 1, 101],
 [2, 102, 10, 110],
 [12, 112, 13, 113]])
```

Метод **`reshape()`** повертає її аргумент із зміненою формою, тоді як метод **`resize`**() змінює сам масив:

```
>>>
>>> a.resize((2, 6))
>>> a
array([[ 0, 1, 2, 10, 12, 13],
 [100, 101, 102, 110, 112, 113]])
```

Якщо при операції такої перебудови один із аргументів задається як -1, то він автоматично розраховується відповідно до інших заданих:

```
>>>
>>> a.reshape((3, -1))
array([[ 0, 1, 2, 10],
 [12, 13, 100, 101],
 [102, 110, 112, 113]])
```

Об'єднання масивів

Декілька масивів можуть бути об'єднані вздовж різних осей за допомогою функцій `hstack` і `vstack`.

`hstack()` об'єднує масиви по перших осях,
`vstack()` – За останніми:

```
>>>
>>> a =np.array([[1, 2], [3, 4]])
>>> b =np.array([[5, 6], [7, 8]])
>>> np.vstack((a, b))
array([[1, 2],
 [3, 4],
 [5, 6],
 [7, 8]])
>>> np.hstack((a, b))
array([[1, 2, 5, 6],
 [3, 4, 7, 8]])
```

Функція `column_stack()` поєднує одновимірні масиви як стовпці двовимірного масиву:

```
>>>
>>> np.column_stack((a, b))
array([[1, 2, 5, 6],
 [3, 4, 7, 8]])
```

Аналогічно для рядків є функція `row_stack()`.

```
>>>
>>> np.row_stack((a, b))
array([[1, 2],
 [3, 4],
 [5, 6],
 [7, 8]])
```

Функція `concatenate()` з'єднує масиви вздовж зазначеної осі:

```
np.concatenate((a1, a2, ..., aN), axis=0)
```

Параметри: `a1, a2, ..., aN` – послідовність подібних до масиву об'єктів. Будь-які об'єкти, які можуть бути перетворені в масиви NumPy. Дані об'єкти повинні мати однакову форму (кількість осей), але не розмір вказаної осі

`axis` – ціле число (необов'язковий). Визначає вісь вздовж якої з'єднуються масиви. За замовчуванням `axis=0` відповідає першій осі.

Повертає: `ndarray` – масив NumPy – масив, що складається з зазначених масивів, з'єднаних уздовж зазначеної осі.

Розбиття масиву

Використовуючи `hsplit`() ви можете розбити масив уздовж горизонтальної осі, вказавши або кількість масивів, що повертаються, однакової форми, або номери стовпців, після яких масив розрізається «ножицями»:

```
>>>
```

```
>>> a =np.arange(12).reshape((2, 6))
>>> a
array([[ 0, 1, 2, 3, 4, 5],
 [6, 7, 8, 9, 10, 11]])
>>> np.hsplit(a, 3) # Розбити на 3 частини
[array([[0, 1], [6, 7]]),
array([[2, 3], [8, 9]]),
array([[ 4, 5], [10, 11]])]
>>> np.hsplit(a, (3, 4)) # Розрізати a після третього
 та четвертого стовпця
[array([[0, 1, 2], [6, 7, 8]]),
array([[3], [9]]),
array([[ 4, 5], [10, 11]])]
```

Функція

`vsplit()` розбиває масив уздовж вертикальної осі, а

`array_split()` дозволяє вказати осі, вздовж яких станеться розбиття.

Копії та уявлення

При роботі з масивами їхні дані іноді необхідно копіювати в інший масив, а іноді ні. Це часто є джерелом плутанини. Можливо 3 випадки:

Взагалі жодних копій.

Просте присвоєння не створює копії масиву, ні копії його даних:

```
>>>
>>> a =np.arange(12)
>>> b = a # Нового об'єкта створено не було
>>> b is a # a і b це два імені для одного і того ж об'єкта
ndarray
True
>>> b.shape = (3,4) # змінить форму a
>>> a.shape
(3, 4)
```

Python передає об'єкти, що змінюються, як посилання, тому виклики функцій також не створюють копій.

Подання або поверхнева копія

Різні об'єкти масивів можуть використовувати ті самі дані. Метод `view()` створює новий об'єкт масиву, що є уявленням тих самих даних.

```
>>>
>>> c = a.view()
>>> c is a
False
>>> c.base is a # c це представлення даних, що належать a
True
>>> c.flags own data
False
>>>
>>> c.shape = (2,6) # форма a не зміниться
>>> a.shape
```

```
(3, 4)
>>> c[0,4] = 1234 # дані a зміняться
>>> a
array([[ 0, 1, 2, 3],
 [1234, 5, 6, 7],
 [8, 9, 10, 11]])
```

Зріз масиву це уявлення:

```
>>>
>>> s = a[:,1:3]
>>> s[:] = 10
>>> a
array([[ 0, 10, 10, 3],
 [1234, 10, 10, 7],
 [8, 10, 10, 11]])
```

Глибока копія

Метод `copy()` створить справжню копію масиву та його даних:

```
>>>
>>> d = a.copy() # створюється новий об'єкт масиву з
                                                    новими даними
>>> d is a
False
>>> d.base is a # d не має нічого спільного з
False
>>> d[0, 0] = 9999
>>> a
array([[ 0, 10, 10, 3],
 [1234, 10, 10, 7],
 [8, 10, 10, 11]])
```

NumPy випадкові числа

Розглянемо, як створювати масиви з випадкових елементів і як працювати з випадковими елементами `NumPy`.

Створювати списки, використовуючи вбудований модуль `random`, а потім перетворювати їх на `numpy.array`:

```
>>>
>>> import numpy as np
>>> import random

>>> np.array([random.random() for i in range(10)])

array([0.99538667, 0.16860511, 0.78952804,
 0.09676316, 0.86110208,
 0.89674666, 0.56401347, 0.63431468,
 0.51110935, 0.64944844])
```

Але є спосіб кращий.

Модуль numpy.random

Створення масивів із випадковими елементами.

Для створення масивів із випадковими елементами служить модуль `numpy.random`.

Найпростіший спосіб задати масив з випадковими елементами – використовувати функцію `sample` (або `random`, або `random_sample`, або `ranf` – це все та сама функція).

```
>>>
>>> np.random.sample()
0.6336371838734877
>>> np.random.sample(3)
array([0.53478558, 0.1441317, 0.15711313])
>>> np.random.sample((2, 3))
array([[ 0.12915769, 0.09448946, 0.58778985],
 [0.45488207, 0.19335243, 0.22129977]])
```

Без аргументів повертає просто число у проміжку [0, 1),

– з одним цілим числом – одномірний масив,

– з кортежем

– масив із розмірами, зазначеними у кортежі (всі числа – із проміжку [0, 1)).

За допомогою функції `randint` або `random_integers` можна створити масив цілих чисел.

Аргументи: `low`, `high`, `size`: від якого, до якого числа (`randint` не включає це число, а `random_integers` включає), і `size` – розміри масиву.

>>>

```
>>> np.random.randint(0,3, 10)
array([0, 2, 0, 1, 1, 0, 2, 2, 2, 0])
>>> np.random.random_integers(0,3, 10)
array([2, 2, 3, 3, 1, 1, 0, 2, 3, 2])
>>> np.random.randint(0,3, (2, 10))
array([[0, 1, 2, 0, 0, 0, 1, 1, 1, 2],
[0, 0, 2, 2, 2, 0, 1, 2, 2, 1]])
```

Також можна генерувати числа згідно з різними розподілами (Гаусса, Парето та інші). Найчастіше потрібен рівномірний розподіл, який можна отримати за допомогою функції `uniform`.

```
>>>
>>> np.random.uniform(2, 8, (2, 10))
array([[ 3.1517914 , 3.10313483, 2.84007134,
 3.21556436, 4.64531786,
 2.99232714, 7.03064897, 4.38691765,
 5.27488548, 2.63472454],
 [6.39470358, 5.63084131, 4.69996748,
 7.07260546, 7.44340813,
 4.10722203, 7.52956646, 4.8596943,
 3.97923973, 5.64505363]])
```

Вибір та перемішування

Перемішати масив NumPy можна за допомогою функції `shuffle`:

```
>>>
>>> a =np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.random.shuffle(a)
>>> a
array([2, 8, 7, 3, 5, 0, 4, 9, 1, 6])
```

Також можна перемішати масив за допомогою функції `permutation` (вона, на відміну `shuffle`, повертає перемішаний масив). Також вона, викликана одним аргументом (цілим числом), повертає перемішану послідовність від 0 до N.

```
>>>
>>> np.random.permutation(10)
array([1, 2, 3, 8, 7, 9, 4, 6, 5, 0])
```

Зробити довільну вибірку з масиву можна за допомогою функції `choice`.

`numpy.random.choice(a, size=None, replace=True, p=None)`

- a: одновимірний масив або число. Якщо масив, проводитиметься вибірка з нього. Якщо число, то вибірка буде з `np.arange(a)` .
- `size`: розмірності масиву. Якщо `None` повертається одне значення.
- `replace` : якщо True, одне значення може вибиратися більше разу.
- `p`: ймовірності. Це означає, що елементи можна вибирати з нерівними можливостями. Якщо не задано, використовується рівномірний розподіл.

```
>>>
>>> a =np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.random.choice(a, 10, p=[0.5, 0.25, 0.25, 0, 0, 0, 0, 0, 0,
0])
array([0, 0, 0, 0, 1, 2, 0, 0, 1, 1])
```

Ініціалізація генератора випадкових чисел.

`seed(число)` – Ініціалізація генератора.

```
>>>
>>> np.random.seed(1000)
>>> np.random.random(10)
array([0.65358959, 0.11500694, 0.95028286,
 0.4821914, 0.87247454,
 0.21233268, 0.04070962, 0.39719446,
 0.2331322, 0.84174072])
>>> np.random.seed(1000)
>>> np.random.random(10)
array([0.65358959, 0.11500694, 0.95028286,
 0.4821914, 0.87247454,
 0.21233268, 0.04070962, 0.39719446,
 0.2331322, 0.84174072])
```

`get_state i set_state` – повертають та встановлюють стан генератора.

```
>>>
>>> np.random.seed(1000)
>>> state =np.random.get_state()
>>> np.random.random(10)
array([0.65358959, 0.11500694, 0.95028286,
 0.4821914, 0.87247454,
 0.21233268, 0.04070962, 0.39719446,
 0.2331322, 0.84174072])
>>> np.random.set_state(state)
>>> np.random.random(10)
array([0.65358959, 0.11500694, 0.95028286,
 0.4821914, 0.87247454,
 0.21233268, 0.04070962, 0.39719446,
 0.2331322, 0.84174072])
```

Деякі корисні функції при роботі з масивами

```
numpy.min(a, axis = None, out = None),
numpy.max(a, axis = None, out = None)
```

повертає мінімальне, максимальне значення елементів масиву відповідно:

```
>>> import numpy as np
>>> print np.min(np.array([ [1.0, -0.5, 3.0], [4.0, 3.0, -
0.5] ]))
-0.5
```

`axis` – опціональний аргумент, індекс осі (вимірювання) масиву яким проводиться пошук мінімального, максимального значення. Під індексом осі розуміється індекс у кортежі `ndarray.shape`.

```
>>> a = np.array([ [ [1.0, 2.0, 3.0], [4.0, 5.0, 6.0] ],\ [ [7.0,
8.0, 9.0], [11, 12, 13] ] ]))
>>> print a
[[[ 1. 2. 3.]
 [4. 5. 6.]]

 [[ 7. 8. 9.]
 [ 11. 12. 13.]]]
>>> print a.shape
(2, 2, 3)

>>> print np.min(a, axis = 0)
[[ 1. 2. 3.]
 [4. 5. 6.]]

>>> print np.min(a, axis = 1)
[[ 1. 2. 3.]
 [7. 8. 9.]]

>>> print np.min(a, axis = 2)
[[ 1. 4.]
 [7. 11.]]
```

`out` – опціональний аргумент, масив, до якого буде поміщений результат. Структура масиву out має відповідати структурі масиву результату. Якщо out = None (за замовчуванням) буде створено новий масив.

`numpy.argmin(a, axis = None),`
`numpy.argmax(a, axis = None)` –

повертає індекс мінімального, максимального значення елементів масиву відповідно:

```
>>> print np.argmin(np.array([ [1.0, -0.5, 3.0],
                                  [4.0, 3.0, -0.5] ]))
1
>>> print np.argmin(a, axis = 0)
[[0 0 0]
 [0 0 0]]
>>> print np.argmin(a, axis = 1)
[[0 0 0]
 [0 0 0]]
>>> print np.argmin(a, axis = 2)
[[0 0]
 [0 0]]
```

`numpy.sum(a, axis = None, dtype = None, out = None),`
`numpy.prod(a, axis = None, dtype = None, out = None)`

повертає суму, добуток елементів масиву відповідно:

```
>>> print np.sum(np.array([ [1.0, -0.5, 3.0], [4.0, 3.0, -
0.5] ]))
10.0
>>> print np.sum(a, axis = 0)
[[ 8. 10. 12.]
 [15. 17. 19.]]
>>> print np.sum(a, axis = 1)
[[ 5. 7. 9.]
 [18. 20. 22.]]
>>> print np.sum(a, axis = 2)
[[ 6. 15.]
 [24. 36.]]
```

`linalg` деякі операції лінійної алгебри

Розглянемо модуль `numpy.linalg`, що дозволяє робити багато операцій із лінійної алгебри.

Зведення у ступінь.

`linalg.matrix_power(M, n)`– Зводить матрицю в ступінь n.

Розкладання.

`linalg.cholesky(a)` – розкладання Холецького.

`linalg.qr(a [, mode])` – QR розкладання.

`linalg.svd(a[, full_matrices, compute_uv])` – сингулярне розкладання.

Деякі характеристики матриць.

`linalg.eig(a)` – власні значення та власні вектори.

`linalg.norm(x[, ord, axis])` – норма вектора чи оператора.

`linalg.cond(x[, p])` – число обумовленості.

`linalg.det(a)` – визначник.

`linalg.slogdet(a)` – знак і логарифм визначника (для уникнення переповнення, якщо сам визначник дуже маленький).

Системи рівнянь

`linalg.solve(a, b)` – вирішує систему лінійних рівнянь `Ax = b`.

`linalg.tensorsolve(a, b[, axes])` – вирішує тензорну систему лінійних рівнянь `Ax = b`.

`linalg.lstsq(a, b [, rcond])` – метод найменших квадратів.

`linalg.inv(a)` – зворотна матриця.

Зауваження:

- `linalg.LinAlgError` – виняток, що викликається цими функціями у разі невдачі (наприклад, при спробі взяти зворотну матрицю від виродженої).
- масиви більшої розмірності у більшості функцій `linalg` інтерпретуються як набір кількох масивів потрібної розмірності. Таким чином, можна одним викликом функції виконувати операції над кількома об'єктами.

```
>>>
>>> a =np.arange(18).reshape((2,3,3))
>>> a
array([[[ 0, 1, 2],
 [3, 4, 5],
 [6, 7, 8]],

 [[ 9, 10, 11],
 [12, 13, 14],
 [15, 16, 17]]])

>>> np.linalg.det(a)
array([0., 0.])
```

Винятки `numpy.linalg.LinAlgError`

```
numpy.linalg.LinAlgError
exception numpy.linalg.LinAlgError
```

Об'єкт `linalg.LinAlgError` генерує винятки Python, викликані функціями модуля `linalg`.

Цей об'єкт утворений класом винятків загального призначення Python і є похідним від нього. Цей клас викликається, тільки якщо подальша робота будь-якої функції модуля `linalg` неможлива.

Приклади

```
>>> import numpy as np
>>> від numpy import linalg as LA
>>>
>>> a = [[0, 0], [0, 0]]
>>>
>>> try:
...    LA.inv(a)
... except LA.LinAlgError:
..   . print('Матриця "a" є або виродженою або прямокутною')
...
```

Матриця "a" є або виродженою або прямокутною

Приклади

Добуток одновимірних масивів є скалярним добутком векторів:

```
>>> a = np.array([1,2])
>>> b = np.array([3,4])
>>>
>>> np.dot(a,b)
11
```

Добуток двовимірних масивів за правилами лінійної алгебри також можливий:

```
>>> a = np.arange(2,6).reshape(2,2)
>>> a
array([[2,3],
[4, 5]])
>>>
>>> b = np.arange(6,10).reshape(2,2)
>>> b
array([[6,7],
[8, 9]])
>>>
>>>np.dot(a,b)
array([[36,41],
[64, 73]])
```

У цьому розміри матриць (масивів) мали бути зацікавленими або рівні, а самі матриці квадратними, або узгодженими, тобто. якщо розміри матриці A дорівнюють [m,k], то розміри матриці повинні бути рівні [k, n]:

```
>>> a = np.arange(2,8).reshape(2,3)
>>> a
array([[2,3, 4],
[5, 6, 7]])
>>>
>>> b = np.arange(4,10).reshape(3,2)
>>> b
array([[4,5],
[6,7],
[8, 9]])
>>>
>>>np.dot(a,b)
array([[58, 67],
```

```
[112, 130]])
```

Також за правилами множення матриць, ми можемо помножити матрицю на вектор (одномірний масив). При цьому в такому множенні вектор стовпець повинен бути праворуч, а вектор рядок зліва:

```
>>> a = np.array([1,2,3])
>>> a
array([1, 2, 3])
>>>
>>> b = np.arange(4,10).reshape(3,2)
>>> b
array([[4,5],
[6,7],
[8, 9]])
>>>
>>>np.dot(a,b)
array([40, 46])
>>>
>>> a = np.arange(1,3).reshape(2,1)
>>> a
array([[1],
[2]]))
>>>
>>> b = np.arange(4,10).reshape(3,2)
>>> b
array([[4,5],
[6,7],
[8, 9]])
>>>
>>>np.dot(b,a)
array([[14],
[20],
[26]]))
```

Квадратні матриці можна зводити до ступеня n тобто. множити самі на себе n разів:

```
>>> a = np.arange(1,5).reshape(2,2)
>>> a
array([[1,2],
[3, 4]])
>>>
>>>np.dot(a,a) # Рівносильно a**2
array([[7, 10],
[15, 22]])
>>>
>>>np.linalg.matrix_power(a,2)
array([[7, 10],
[15, 22]])
>>>
>>>np.linalg.matrix_power(a,5)
array([[1069,1558],
[2337, 3406]])
>>>
>>>np.linalg.matrix_power(a,0)
```

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

Досить часто доводиться обчислювати ранг матриць:

```
>>> a = np.arange(1,10).reshape(3,3)
>>> a
array([[1,2, 3],
[4,5, 6],
[7, 8, 9]])
>>>
>>> np.linalg.matrix_rank(a)
2
>>>
>>> b = np.arange(1,24,2).reshape(3,4)
>>> b
array([[1, 3, 5, 7],
[9, 11, 13, 15],
[17, 19, 21, 23]])
>>>
>>> np.linalg.matrix_rank(b)
2
```

Ще частіше доводиться обчислювати визначник матриць, хоча результат вас може трохи здивувати:

```
>>> a = np.array([[1,3],[4,3]])
>>> a
array([[1,3],
[4, 3]])
>>>
>>> np.linalg.det(a)
-8.999999999999982
>>>
>>> 1*3 - 3*4 # Результат має бути цілим числом
-9
```

У даному випадку, через двійкову арифметику, результат не ціле число і округляти до найближчого цілого доведеться вручну. Це з тим, що алгоритм обчислення визначника використовує LU-розкладання – це набагато швидше ніж звичайний алгоритм, але за швидкість все ж таки доводиться трохи заплатити ручним округленням (звичайно, якщо таке потрібно):

```
>>> np.linalg.det(a)
-8.999999999999982
>>> round(np.linalg.det(a))
-9.0
>>>
>>> b = np.arange(1,48,3).reshape(4,4)
>>> np.linalg.det(b)
-1.0223637331664275e-27
>>> round(np.linalg.det(b))
-0.0
```

Транспонування матриць:

```
>>> a
matrix([[1,3],
[4, 3]])
```

```
>>>
>>>a.T    # або = np.transpose(a)

matrix([[1,4],
[3, 3]])
>>>
>>> b
array([[1, 4, 7, 10],
[13,16, 19, 22],
[25,28, 31, 34],
[37, 40, 43, 46]])
>>>
>>>b.T
array([[1, 13, 25, 37],
[4, 16, 28, 40],
[7, 19, 31, 43],
[10, 22, 34, 46]])
```

Обчислення зворотних матриць:

```
>>> a = np.array([[1,3],[4,3]])
>>> a
array([[1,3],
[4, 3]])
>>>
>>> b = np.linalg.inv(a)
>>> b
array([[-0.33333333,0.33333333],
[0.44444444, -0.11111111]])
>>>
>>>np.dot(a,b)
array([[1., 0.],
       [0, 1.]])
```

Розв'язання систем лінійних рівнянь:

```
... # система з двох лінійних рівнянь:
...
... # 1*x1 + 5*x2 = 11
... # 2*x1 + 3*x2 = 8
...
>>> a = np.array([[1,5],[2,3]])
>>> b = np.array([11,8])
>>>
>>> x = np.linalg.solve(a,b)
>>> x
array([ 1., 2.])
>>>
>>>np.dot(a,x)
array ([11., 8.])
```

Перетворення Фур'є

По-простому, перетворення Фур'є – розкладання деякого сигналу на гармонійні (синуси або косинуси) коливання (спектр).

Перетворення Фур'є є інтегральним перетворенням. Якщо йдеться про дискретний сигнал, то інтеграл перетворюється на суму (і стає дискретним перетворенням Фур'є, ДПФ). Щоб порахувати таку суму N елементів, треба здійснити N^2 операцій із комплексними числами. Але досить давно (Cooley, Tukey, 1965 р, а ще сам Гаус в 1805 р.) придумав алгоритм, що обчислює ДПФ N елементів у N * log (N) операцій (більшість з яких над дійсними числами), що суттєво економить обчислювальний час. Такий алгоритм часто називають "швидке перетворення Фур'є, ШПФ (Fast Fourier Transform, FFT)". Саме так реалізовано ДПФ у сучасних комп'ютерних програмах.

У бібліотеці `NumPy` міститься все, що потрібно для дискретного перетворення Фур'є. Все це лежить у модулі `numpy.fft`.

Ось ці функції.

Загальний випадок: сигнал може бути як із дійсних чисел, так і з комплексних.

`fft(a, n=None, axis=-1)` – пряме одновимірне ДПФ.

`ifft(a, n=None, axis=-1)` – зворотне одномірне ДПФ.

тут:

`a` – "сигнал", вхідний масив (масив numpy array або навіть пітонівський список або кортеж, якщо в ньому лише числа).

Масив може бути і багатовимірним, тоді обчислюватиметься багато одномірних ПФ за рядками (за замовчуванням) або стовпцями, залежно від параметра `axis`.

Наприклад, `a` – двомірний, `a [n] [m]`:
при `axis=1` чи -1 буде таке (`під fourier(a...)` розуміється результат дії ПФ на a):

```
[fourier(a[0][j]), fourier(a[1][j]), ...
                                     foirier(a[n][j])]
```

При `axis=0` таке:

```
[fourier(a[i][0]), fourier(a[i][1]),...
                                     foirier(a[i][m])]
```

`n`– Скільки елементів масиву брати. Якщо менше довжини масиву, то обрізати, якщо більше, доповнити нулями, за замовчуванням `len(a)`.

`fft2(a, s=None, axes=(-2, -1))` – пряме двовимірне ПФ.

`ifft2(a, s=None, axes=(-2, -1))` – зворотне двовимірне ПФ.

`fftn(a, s=None, axes=None)` – пряме багатовимірне ПФ.

`ifftn(a, s=None, axes=None)` – зворотне багатовимірне ПФ.

Так само, як і для одномірних, але s і axes тепер кортежі для кожної розмірності. Про розмірність fftn, ifftn здогадаються за розмірністю вхідних масивів або s і axes.

Коли сигнал дійсний (real) (мабуть, найпоширеніший випадок):

`rfft(a, n=None, axis=-1)` – пряме одновимірне ДПФ (для дійсних чисел).

`irfft(a, n=None, axis=-1)` – зворотне одномірне ДПФ.

`rfft2(a, s=None, axes=(-2, -1))` – пряме двовимірне ДПФ.
`irfft2(a, s=None, axes=(-2, -1))` – зворотне двомірне ДПФ.
`rfftn(a, s=None, axes=None)` – пряме багатовимірне ДПФ.
`irfftn(a, s=None, axes=None)` – зворотне багатовимірне ДПФ.
Так само, як і для загального випадку.

Всі ці функції повертають масив відповідної розмірності, в якому записано результат ДПФ.

Якщо довжина вхідного масиву (або будь-якої його розмірності) N, то загальному випадку (з комплексним сигналом) довжина вихідного масиву N.

Там містяться спочатку позитивні частоти від нуля до частоти Котельникова – Найквіста, потім негативні у порядку зростання.

У разі дійсного сигналу негативні частоти повністю симетричні позитивним, і тоді немає потреби їх записувати: довжина вихідного масиву N/2+1, частоти від нуля до частоти Котельникова.

Якщо спектр сигналу дійсний (а сигнал має "ермітову симетрію": його половини симетричні щодо центру за модулем і є комплексно пов'язаними один одному), то можна застосувати такі функції:

`hfft(a, n=None, axis=-1)` – пряме одновимірне ДПФ.
`ihfft(a, n=None, axis=-1)` – зворотне одномірне ДПФ.
Довжина вхідного масиву N, а вихідного 2*N+1.

Крім того, є допоміжні функції (буде зрозумілішим з прикладу):
`fftfreq(n, d=1.0)` – повертає частоти для вихідних масивів функцій fft*.
`rfftfreq(n, d=1.0)` – повертає частоти для вихідних масивів функцій rfft*.
Тут – `n` – довжина вхідного масиву, `d` – період дискретизації (зворотна частота дискретизації).
`fftshift(x, axes=None)` – перетворює масив (з результатом ДПФ, від функцій `fft*`) так, щоб нульова частота була в центрі.
`ifftshift(x, axes=None)` – здійснює зворотну операцію.

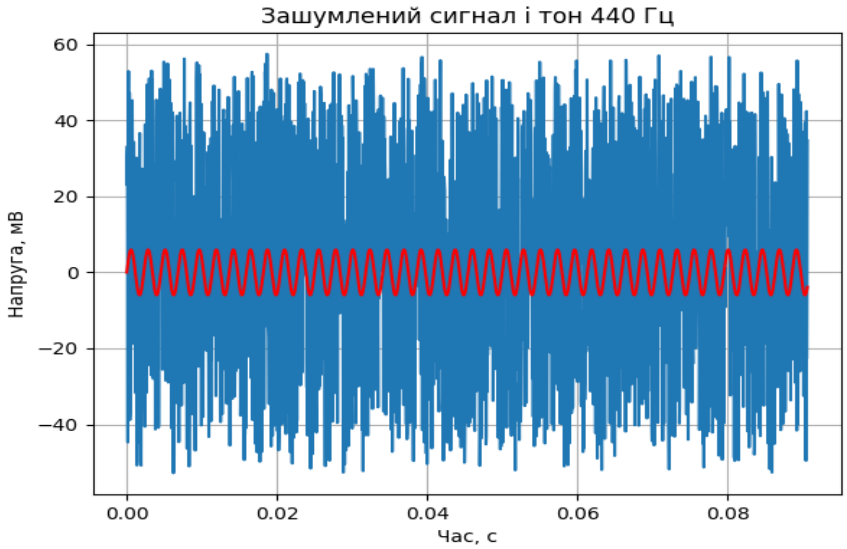
Наведемо такий приклад. Припустимо, мікрофоном записано якийсь шум, і треба визначити, чи є там якийсь тон.

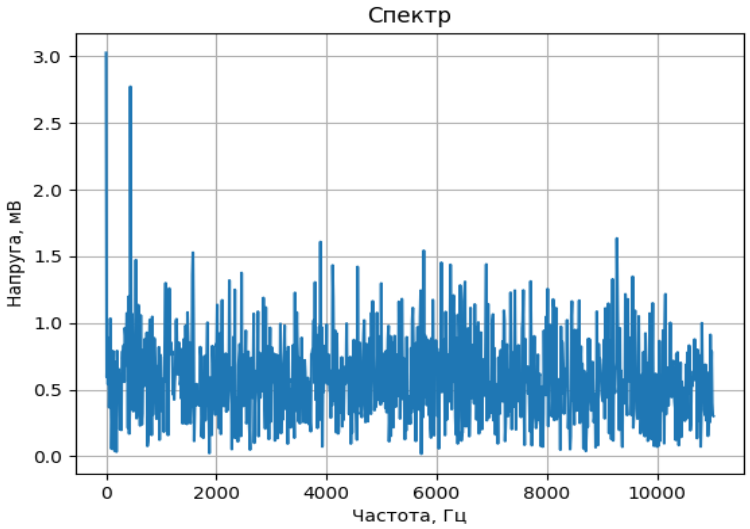
```
from numpy import array, arange, abs as np_abs
from numpy.fft import rfft, rfftfreq

from numpy.random import uniform
from math import sin, pi
import matplotlib.pyplot as plt

# а можна імпортувати numpy та писати: numpy.fft.rfft

FD = 22050 #частота дискретизації, відліків за секунду
```

```
     # а це означає, що в дискретному сигналі
     # представлені частоти від нуля до 11025 Гц
     # (це і є теорема Котельникова)

N = 2000 # Довжина вхідного масиву, 0.091 секунд при
         # такій частоті дискретизації
# згенеруємо сигнал із частотою 440 Гц довжиною N
pure_sig =  \
 array([6.*sin(2.*pi*440.0*t/FD) for t in range(N)])

# згенеруємо шум, теж довжиною N (це важливо!)

noise = uniform(-50.,50., N)

#підсумовуємо їх і додамо постійну складову 2 мВ #(припустимо, не
дуже хороший мікрофон попався.
# Або звукова карта або АЦП)
sig = pure_sig + noise + 2.0

# у numpy так перевантажена функція додавання

# Обчислюємо перетворення Фур'є.
# Сигнал дійсний, тому треба
# використовувати rfft, це швидше, ніж fft

spectrum = rfft(sig)
# намалюємо все це, використовуючи matplotlib
# Спочатку сигнал зашумлений і тон окремо
plt.plot(arange(N)/float(FD), sig) # по осі часу секунди!
plt.plot(arange(N)/float(FD), pure_sig, 'r') # Чистий
                    # сигнал буде намальований червоним

plt.xlabel(u'Час, с')
plt.ylabel(u'Напруга, мВ')
plt.title(u'Зашумлений сигнал і тон 440 Гц')

plt.grid(True)
plt.show()

# коли закриється цей графік, відкриється наступний
#           Потім спектр
plt.plot(rfftfreq(N, 1./FD), np_abs(spectrum)/N)
# rfftfreq зробить всю роботу з перетворення
#          номерів елементів масиву в герці
# нас цікавить тільки спектр амплітуд, тому
#             використовуємо abs з numpy
             # (діє на масиви поелементно)

# ділимо на число елементів, щоб амплітуди були в
# мілівольтах, а не в сумах Фур'є.
# Перевірити просто - постійні складові повинні
# збігатися в згенерованому сигналі та в спектрі
```

```
plt.xlabel(u'Частота, Гц')
plt.ylabel(u'Напруга, мВ')
plt.title(u'Спектр')
plt.grid(True)
plt.show()
```

Результат виглядає таким чином:





Питання для самоконтролю до теми 2

1. Як створити масив у NumPy?
2. Які базові операції з масивами є у бібліотеці NumPy?
3. Які основні математичні функції реалізовані у бібліотеці NumPy?
4. Як створити масив випадкових чисел у NumPy?
5. Які операції лінійної алгебри реалізовані у бібліотеці NumPy?

Завдання до теми 2

Напишіть Python-скрипт, який виконує наступні кроки: 1) Імпортує бібліотеку `NumPy` під псевдонімом `np`. 2) Створює двовимірний масив A розміром `3x4` з нулів, використовуючи функцію `np.zeros()`. 3) Перевіряє і виводить на екран три ключові атрибути масиву A: його розмірність (`ndim`), форму (`shape`) та загальну кількість елементів (`size`). 4) Створює одномірний масив B з послідовності чисел від 5 до 10 (не включно) за допомогою `np.arange()`. 5) Виконує та виводить результат по елементного піднесення масиву B до квадрату, демонструючи базову математичну операцію.

3. БІБЛІОТЕКА MATPLOTLIB

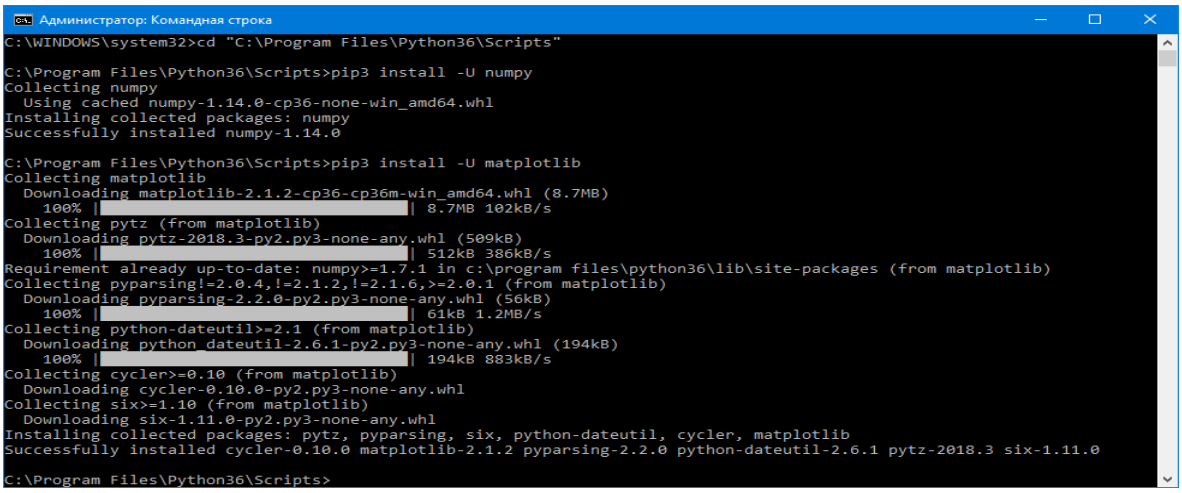
`matplotlib` – набір додаткових модулів (бібліотек) мови Python. Надає засоби для побудови найрізноманітніших 2D графіків та діаграм даних. Переваги цієї бібліотеки – простота використання. Для побудови дуже складних і барвисто оформлених діаграм достатньо кількох рядків коду. При цьому якість зображень більш ніж достатньо для їх опублікування. Робота цього модуля зазвичай має на увазі також використання модуля NumPy.

Встановлення

У Windows установка і matplotlib і NumPy не повинні викликати жодних проблем. Використовуємо pip – систему керування пакетами, яка використовується для встановлення та керування програмними пакетами, написаними на Python.

Якщо Python, наприклад, встановлений у C:\Program Files\Python36\, то в командному рядку адміністратора (це важливо) необхідно виконати такі команди:

```
python -m pip install numpy
python -m pip install matplotlib
```



Протокол роботи:

```
C:\WINDOWS\system32>cd "C:\Program Files\Python36\Scripts"

C:\Program Files\Python36\Scripts>pip3 install -U numpy
Collecting numpy
 Using cached numpy-1.14.0-cp36-none-win_amd64.whl
Installing collected packages: numpy
Successfully installed numpy-1.14.0

C:\Program Files\Python36\Scripts>pip3 install -U matplotlib
Collecting matplotlib
 Downloading matplotlib-2.1.2-cp36-cp36m-win_amd64.whl (8.7MB)
 100% |████████████████████████████████| 8.7MB 102kB/s
Collecting pytz (від matplotlib)
 Downloading pytz-2018.3-py2.py3-none-any.whl (509kB)
 100% |████████████████████████████████| 512kB 386kB/s
```

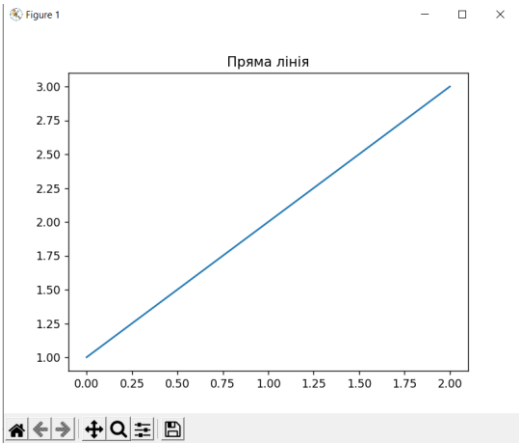
```
Requirement already up-to-date: numpy>=1.7.1 in c:\program
files\python36\lib\site-packages (from matplotlib)
Collecting pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 (from matplotlib)
 Downloading pyparsing-2.2.0-py2.py3-none-any.whl (56kB)
 100% |████████████████████████████████| 61kB 1.2MB/s
Collecting python-dateutil>=2.1 (from matplotlib)
 Downloading python_dateutil-2.6.1-py2.py3-none-any.whl (194kB)
 100% |████████████████████████████████| 194kB 883kB/s
Collecting cycler>=0.10 (від matplotlib)
 Downloading cycler-0.10.0-py2.py3-none-any.whl
Collecting six>=1.10 (from matplotlib)
 Downloading six-1.11.0-py2.py3-none-any.whl
Installing collected packages: pytz, pyparsing, six, python-dateutil,
cycler, matplotlib
Successfully installed cycler-0.10.0 matplotlib-2.1.2 pyparsing-2.2.0
python-dateutil-2.6.1 pytz-2018.3 six-1.11.0

C:\Program Files\Python36\Scripts>
```

Перевірка.

Після встановлення перевіряємо працездатність. Запускаємо консоль Python, вводимо:

```
>>> import matplotlib as mpl
>>> print ('Current version on matplotlib library is',
mpl.__version__)
Current version on matplotlib library is 2.1.2

# де-факто стандарт виклику pyplot у python
import matplotlib.pyplot as plt
plt.plot([1,2,3])
[<matplotlib.lines.Line2D object at 0x0301E810>]

plt.title('Пряма лінія')
<matplotlib.text.Text object at 0x03062430>

plt.show()
```

В результаті має з'явитися вікно діаграми:



Що було зроблено?

З пакету `matplotlib` імпортований модуль `pyplot` під назвою `plt`.

Модуль `pyplot` містить функції (схожі на команди) створення діаграм та зміни властивостей їх елементів.

Функція `plot()` будує прямокутні двовимірні діаграми (графіки) координатах X – Y.

Якщо функції plot передано один аргумент (у нашому випадку – список [1,2,3]), вона розглядає його як сукупність значень відкладаються по осі Y, тоді по осі X йому відповідатиме автоматично згенерований набір чисел 0,1,2. .N – 1, де N – число елементів у переданому списку.

Функція title() задає заголовок діаграми, а функція `show()` виводить інтерактивне вікно діаграми. Загалом, все дуже просто.

Налаштування

Matplotlib можна легко налаштувати через конфігураційний файл matplotlibrc. Якщо Python встановлений в папку C:\Program Files\Python36\, то настроювальний файл розташовується в C:\Program Files\Python36\Lib\site-packages\matplotlib\mpl-data\.

Вносити зміни можна прямо тут, але краще скопіювати файл до папки користувача, наприклад: C:\Documents and Settings\UserName\.matplotlib\, інакше при переустановленні пакета конфігураційний файл буде перезаписано. Параметри пакета задаються у файлі як пар властивість : значення, символ # відокремлює коментар.

Властивість `font.family` визначає тип активного шрифту за умовчанням, може набувати п'ять значень: `serif`, `sans-serif`, `cursive`, `fantasy`, `monospace`.

У свою чергу властивості `font.serif`, `font.sans-serif`, `font.cursive`, `font.fantasy`, `font.monospace` містять списки імен шрифтів (через кому, в порядку зменшення пріоритету), що відповідають кожному типу.

Наприклад, для "русифікації" `matplotlib`, можливо, доведеться змінити імена шрифтів у списку, на імена доступних в системі шрифтів, що містять кириличні символи. Наприклад так:

```
font.serif : Verdana, Arial
font.sans-serif : Tahoma, Arial
font.cursive: Courier New, Arial
font.fantasy: Comic Sans MS, Arial
font.monospace : Arial
```

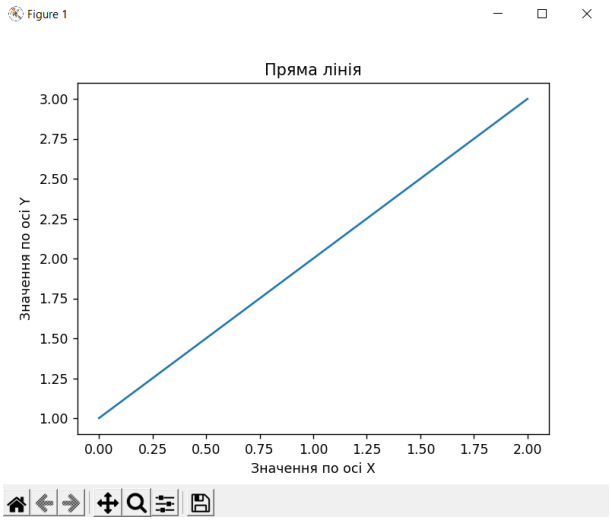
Зауважимо, що останні версії пакета встановлюються з коректним настроювальним файлом.

Розглянемо приклад:

```
>>> plt.plot([1,2,3])
[<matplotlib.lines.Line2D object at 0x000001BB6DFCA588>]
>>> plt.title('Пряма лінія')
Text(0.5,1,'Пряма лінія')
>>> plt.xlabel(u'Значення по осі X')
```

```
Text(0.5,0,'Значення по осі X')
>>> plt.ylabel(u'Значення по осі Y')
Text(0,0.5,'Значення по осі Y')
>>> plt.show()
```

Результат роботи тексту програми:



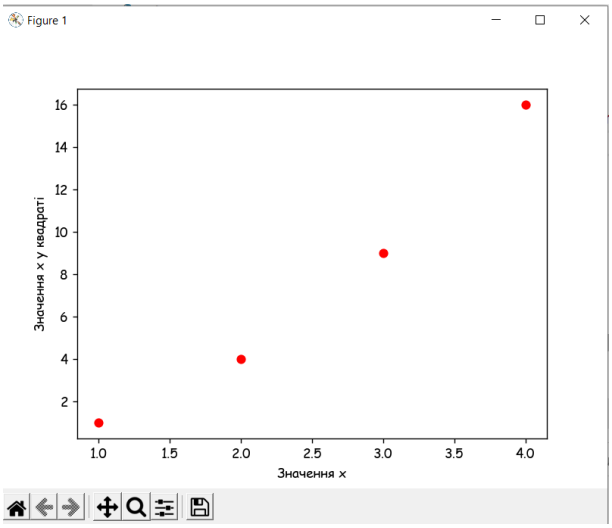
Функції `pyplot.xlabel()`, `pyplot.ylabel()` встановлюють підписи до осей X та Y відповідно.

Пакет `matplotlib` можна налаштовувати динамічно (під час виконання програми). У пакеті визначено структуру `matplotlib.rcParams`, з якою працюють зі словником. Ключові слова в даному випадку – імена властивостей файлу matplotlibrc.

Налаштування шрифтів через `matplotlib.rcParams` можна змінити, наприклад:

```
>>>import matplotlib as mpl
>>>import matplotlib.pyplot as plt
>>> mpl.rcParams['font.family'] = 'fantasy'
>>> mpl.rcParams['font.fantasy'] = 'Comic Sans MS, Arial'
>>> plt.plot([1,2,3,4], [1,4,9,16], 'ro')
[<matplotlib.lines.Line2D object at 0x000001BB6E03B470>]
>>> plt.xlabel('Значення x')
Text(0.5,0,'Значення x')
>>> plt.ylabel('Значення x у квадраті')
Text(0,0.5,'Значення x у квадраті')
>>> plt.show()
```

В результаті маємо отримати:



Функції `pyplot.plot()` можна передати довільну кількість пар аргументів типів `list` або array (тип array визначений у модулі `numpy`).

У прикладі вище передано одну пару: [1,2,3,4], [1,4,9,16]. Перший елемент кожної пари функція розглядає як значення, що відкладаються по осі X, другий елемент як значення осі Y. Відповідно для кожної пари елементів будуватися своя крива на діаграмі.

Після кожної пари даних може бути переданий додатковий аргумент типу string – рядок форматування, який визначає зовнішній вигляд кривої.

Формат рядка запозичений із системи Matlab. Рядок включає три підрядки.

- перший підрядок задає колір графічного елемента,
- друга стиль маркера,
- третій стиль лінії.

У прикладі 'ro' = 'r' + 'o', де 'r' – червоний, 'o' – кружок. За замовчуванням 'b-' – синя безперервна лінія.

Архітектура matplotlib

Одне з основних завдань, що виконує matplotlib – надання набору функцій та інструментів для представлення та управління Figure (так називається основний об'єкт) разом із усіма внутрішніми об'єктами, з якого він складається. Але в matplotlib є інструменти для обробки подій і, наприклад, анімації. Завдяки їм ця бібліотека здатна створювати інтерактивні графіки на основі подій натискання кнопки або руху миші.

Архітектура matplotlib логічно розділена на три шари, розташовані на трьох рівнях. Комунікація непряма – кожен шар може взаємодіяти тільки з тим, що розташований під ним, але не над.

Ось ці шари:

- Шар сценарію
- Художній шар
- Шар бекенда

Шар бекенда

Шар Backend є нижнім на діаграмі з архітектурою усієї бібліотеки. Він містить усі API та набір класів, які відповідають за реалізацію графічних елементів на низькому рівні.

- `FigureCanvas` – Це об'єкт, що втілює область малювання.
- `Renderer`– Об'єкт, який малює по FigureCanvas.
- `Event` – об'єкт, що обробляє введення від користувача (події з клавіатури та миші)

Художній шар

Середнім шаром є художній (artist). Усі елементи, що становлять графік, такі як назва, мітки осей, маркери тощо, є екземплярами цього об'єкта. Кожен їх грає свою роль ієрархічної структурі.

Є два мистецькі класи: примітивний та складовий.

- Примітивний – це об'єкти, які є базовими елементами для формування графічного представлення графіка, наприклад, Line2D, або геометричні фігури, такі як прямокутник коло або навіть текст.
- Складові – об'єкти, що складаються з кількох базових (примітивних). Це осі, шкали та діаграми.

На цьому рівні часто доводиться мати справу з об'єктами, що займають високе становище в ієрархії: графік, система координат, осі. Тому важливо повністю розуміти, яку роль вони відіграють. Нижче наведено три основні художні (складові об'єкти), які часто використовуються на цьому рівні.

- `Figure` – Об'єкт, що займає верхню позицію в ієрархії. Він відповідає всьому графічному уявленню і може містити багато систем координат.
- `Axes` – це цей графік. Кожна система координат належить лише одному об'єкту Figure і має два об'єкти Axis (або три, якщо йдеться про тривимірний графік). Інші об'єкти, такі як назва, мітки x та y, належать окремо осям.
- `Axis` – враховує числові значення в системі координат, визначає межі та керує позначеннями на осях, а також відповідним кожному з них текстом. Положення шкал визначається об'єктом Locator, а зовнішній вигляд – Formatter.

Шар сценарію (pyplot)

Художні класи та пов'язані з ними функції (API matplotlib) підходять усім розробникам, особливо тим, хто працює із серверами веб додатків або розробляє графічні інтерфейси. Але для обчислень, зокрема для аналізу та візуалізації даних найкраще підходить шар сценарію. Він включає інтерфейс pyplot.

`pylab i pyplot`

З погляду користувача існують дві бібліотеки: pylab та pyplot. Pylab – це модуль, що встановлюється разом з matplotlib, а pyplot – внутрішній модуль matplotlib. Обидва часто посилаються в скриптах:

```
for pylab import *
import matplotlib.pyplot as plt
import numpy as np
```

Pylab поєднує функціональність pyplot із можливостями NumPy в одному просторі імен, тому окремо імпортувати NumPy не потрібно. Більше того, при імпорті pylab функції з pyplot та NumPy можна викликати без посилання на модуль (простір імен).

```
plot(x,y)
array([1,2,3,4])
# замість
plt.plot()
np.array([1,2,3,4]
```

Пакет pyplot пропонує класичний інтерфейс Python для програмування, має власний простір має та потребує окремого імпорту NumPy. У наступних матеріалах використається цей підхід. Його ж застосовує більшість програмістів на Python.

Призначення кнопок інтерактивного вікна діаграми

Кнопка  (pan/zoom) призначена для прокручування/масштабування діаграми.

Натискаємо на кнопку, покажчик миші набуває вигляду перехрещених стрілок. Якщо рухати мишу і утримувати ліву кнопку – діаграма прокручуватиметься у напрямку руху покажчика. Якщо рухати мишу і утримувати праву кнопку – масштаб діаграми буде змінюватися, при русі вгору/вправо – зменшуватись, вниз/вліво – збільшуватись.

При натиснутих клавішах "x" та "y" переміщення/масштабування діаграми відбуватиметься по осях X та Y відповідно.

При натиснутій кнопці Ctrl діаграма буде змінюватися так, щоб збереглися її пропорції (aspect ratio).

Кнопка  при натисканні включає режим прямокутного масштабування.

Якщо натиснути кнопку вказівник миші набуде вигляду хреста.

При натиснутій лівій кнопці миші на діаграмі можна виділити прямокутну область. Після звільнення кнопки масштаб діаграми буде змінено так, щоб виділена область зайняла якомога більшу площу діаграми.

Кнопки  дозволяють переміщатися з історії зміни діаграми. Всі зміни, що вносяться в діаграму користувачем (масштабування, прокручування), запам'ятовуються.

Кнопки зі стрілками дозволяють переміщатися вперед/назад за варіантами діаграми. Кнопка з будиночком повертає діаграму до початкового стану.

Кнопка  викликає стандартний системи діалог збереження файла.

Дозволяє зберегти діаграму у файлі. Можна вибрати такі формати діаграми: png, pdf, svg, eps, ps.

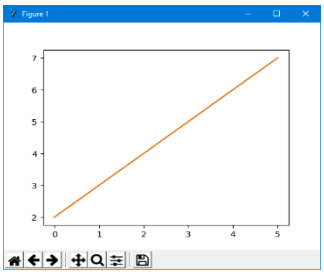
У всіх прикладах імпортуємо модулі:

```
import matplotlib as mpl
# де-факто стандарт виклику pyplot у python
import matplotlib.pyplot as plt
```

Список координат, x координати числа 0, 1, 2, 3...

Список y[2,3,4,5,6,7] координат, x координати утворюють послідовність 0, 1, 2, ...:

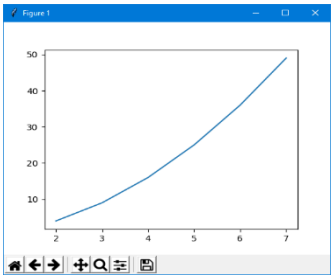
```
plt.plot([2,3,4,5,6,7])
plt.show()
```



Списки x та y координат

Список x та y координат [2,3,4,5,6,7], [4,9,16,25,36,49]):

```
plt.plot([2,3,4,5,6,7],[4,9,16,25,36,49]);plt.show()
```

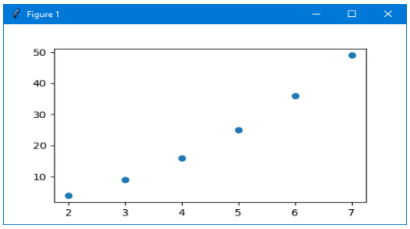


Зображення точок

Список x та y координат – `[2,3,4,5,6,7], [4,9,16,25,36,49]);`

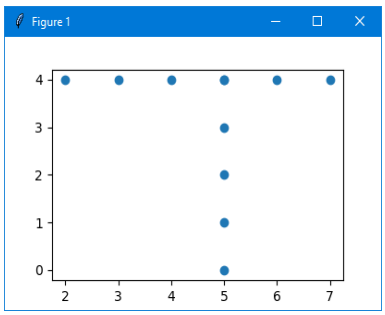
Зображаємо лише точки

```
plt.scatter([2,3,4,5,6,7],[4,9,16,25,36,49]);plt.show()
```



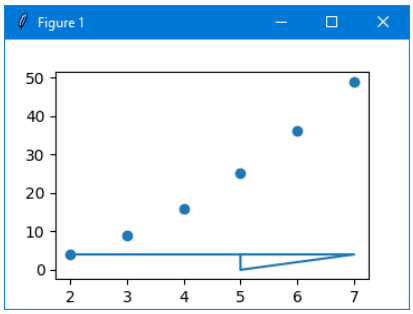
«Довільні» координати

```
plt.scatter([2,3,4,5,6,7,5,5,5,5,5],
                [4,4,4,4,4,4,0,1,2,3,4 ,]);  plt.show()
```



Декілька графіків на одному аркуші

```
plt.scatter([2,3,4,5,6,7],[4,9,16,25,36,49])
plt.plot([2,3,4,5,6,7,5,5,5,5,5],
                              [4,4,4,4,4,4,0,1,2,3,4 ,])

plt.show()
```

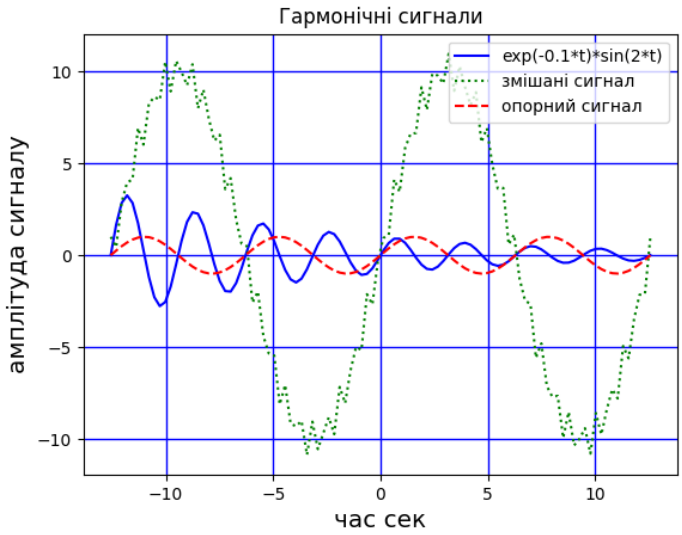


«Прикрашання» та багато графіків на діаграмі

Малюємо та зберігаємо у файлі три графіки на одній діаграмі в одному масштабі (див. коментарі скрипту):

```
import matplotlib.pyplot as plt
import numpy as np
# незалежна змінна
x = np.linspace (-4 * np.pi, 4 * np.pi, 100)
# назва осей
xlab='час сек'
ylab='амплітуда сигналу'
# перший графік
y1= np.sin(2*x)*np.exp(-0.1*x)
# Легенда
leg1='exp(-0.1*t)*sin(2*t)'
# другий графік
y2=10*np.sin(0.5*x)+np.cos(10*x)
leg2='змішані сигнал'
# третій графік
y3=np.sin(x)
leg3='опорний сигнал'
# Включаємо сітку по осі X та осі Y.
# Задаємо колір товщину сітки
plt.grid(color = 'b',linewidth = 1)
```

```
# Задаємо підписи до осей X та осі Y та розмір шрифту
plt.xlabel(xlab, fontsize = 'x-large')
plt.ylabel(ylab, fontsize = 'x-large')
# Задаємо заголовок діаграми
plt.title('Гармонічні сигнали')
# будуємо графіки
plt.plot(x,y1,'b-',label=leg1)
plt.plot(x,y2,'g:',label=leg2)
plt.plot(x,y3,'r--',label=leg3)

# задаємо висновок легенди та її розташування
plt.legend(loc='best')
# Включаємо сітку
plt.grid(True)
# Зберігаємо побудовану діаграму у файл
# Задаємо ім'я файлу та його тип
plt.savefig('signal3.png', format = 'png')
# візуалізуємо графіки
plt.show()
```



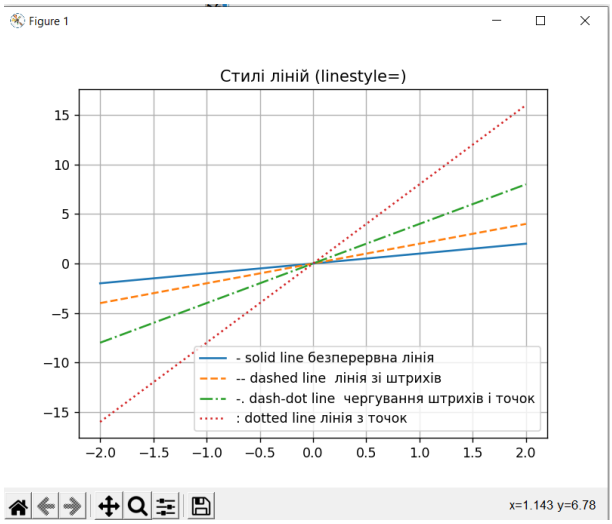
Як задавати стилі ліній, маркери та кольори ліній, можна подивитись у документації або скористатись «демо скриптом»:

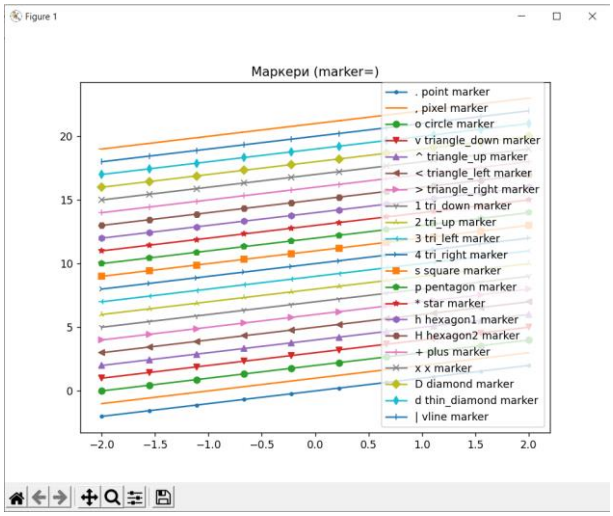
```
import sys
import math
import os
import matplotlib.pyplot as plt
import numpy as np

##sys.exit(0)
import matplotlib as mpl

# Виведення на екран поточної версії бібліотеки
#                                          matplotlib
```

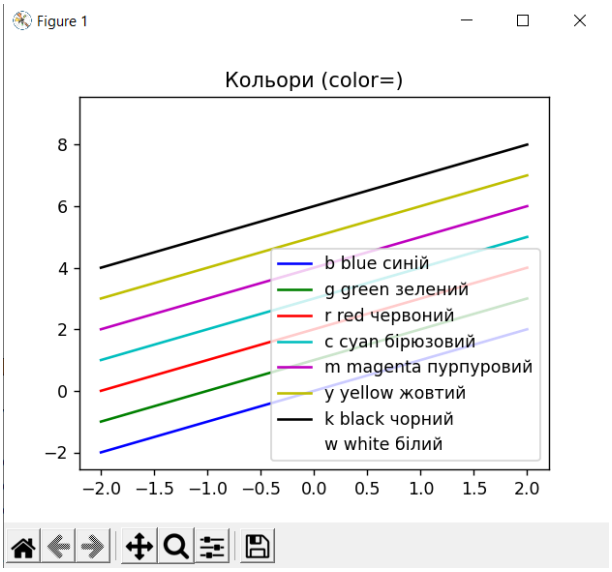
```
print('Current version on matplotlib library is',
                                        mpl.__version__)

x = np.linspace(-2, 2, 10)
y1=1*x
y2=2*x
y3=4*x
y4=8*x

s1= ['-', 'solid line', 'безперервна лінія']
s2= ['--', 'dashed line ', 'лінія зі штрихів']
s3= ['-.', 'dash-dot line ', 'чергування штрихів і
                                                  точок']
s4= [':', 'dotted line', 'лінія з точок']

leg1=s1[0]+' '+s1[1]+' '+s1[2]
leg2=s2[0]+' '+s2[1]+' '+s2[2]
leg3=s3[0]+' '+s3[1]+' '+s3[2]
leg4=s4[0]+' '+s4[1]+' '+s4[2]

ax=plt.subplot(111)
# можна поміняти розміри вікна графіка у граф. вікні
box=ax.get_position()

ax.set_position([box.x0, box.y0, box.width*1.0 ,
                                       box.height*1.0])

p1 = plt.plot (x, y1, linestyle = s1 [0], label = leg1)
p2=plt.plot(x,y2,linestyle=s2[0], label=leg2)
p3 = plt.plot (x, y3, linestyle = s3 [0], label = leg3)
p4 = plt.plot (x, y4, linestyle = s4 [0], label = leg4)

plt.title('Стилі ліній (linestyle=)')
#ax.legend(loc=(1.0,0.5), mode='expand' )
#ax.legend(mode='expand', bbox_to_anchor=(1, 0.05),
                           loc='upper center')
                                          #left best
ax.legend(loc='lower right')
# Включаємо сітку
plt.grid()
plt.show()

#sys.exit(0)
# marker
plt.close()
mar=[
          '.', 'point marker',
          ',', 'pixel marker',
          'o', 'circle marker',
          'v', 'triangle_down marker',
          '^', 'triangle_up marker',
          '<', 'triangle_left marker',
          '>', 'triangle_right marker',
```

```
          '1', 'tri_down marker',
          '2', 'tri_up marker',
          '3', 'tri_left marker',
          '4', 'tri_right marker',
          's', 'square marker',
          'p', 'pentagon marker',
          '*', 'star marker',
          'h', 'hexagon1 marker',
          'H', 'hexagon2 marker',
          '+', 'plus marker',
          'x', 'x marker',
          'D', 'diamond marker',
          'd', 'thin_diamond marker',
          '|', 'vline marker',
          '_', 'hline marker'
]
leg=[]
for i in range(0,len(mar),2):
   leg.append(mar[i]+' '+mar[i+1])
mpl.rcParams['figure.figsize'] = (8.0, 6.0)
kk = int (len (mar) / 2)
for i in range(kk):
    plt.plot(x, x+i, marker=leg[i][0], label=leg[i])
plt.legend(loc='lower right')
plt.title('Маркери (marker=)')
plt.show()

#color
col=[
     'b', 'blue синій',
     'g', 'green зелений',
     'r', 'red червоний',
     'c', 'cyan бірюзовий',
     'm', 'magenta пурпуровий',
     'y', 'yellow жовтий',
     'k', 'black чорний',
     'w', 'white білий'
]

leg=[]
for i in range(0,len(col),2):
    leg.append(col[i]+' '+col[i+1])
mpl.rcParams['figure.figsize'] = (5.0, 4.0)
kk = int (len (col) / 2)
for i in range(kk):
    plt.plot(x, x+i, color=leg[i][0], label=leg[i])
plt.legend(loc='lower right')
plt.title('Кольори (color=)')
plt.show()
```

Демо графіки:





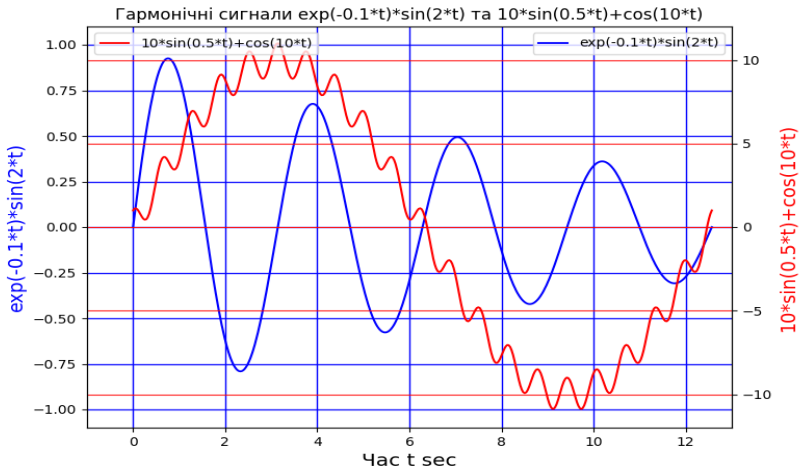


Два графіки на діаграмі в індивідуальних масштабах

```
#Зверніть увагу на «кому» після імені змінної
#           line_01,= ax_01.plot(X, Y_01, 'b-')
import numpy as np
```

```
import matplotlib as mpl
import matplotlib.pyplot as plt
# Значення по осі X час, t
X = np.linspace(0, 4 * np.pi, 400)
# Значення по осях Y_01 & Y_02
leg_01='exp(-0.1*t)*sin(2*t)'
Y_01 = np.sin(2*X)*np.exp(-0.1*X)
# "Модульований" сигнал
leg_02="10*sin(0.5*t)+cos(10*t)"
Y_02=10*np.sin(0.5*X)+np.cos(10*X)
# Задамо розміри зображення діаграми
mpl.rcParams['figure.figsize'] = (8.0, 6.0)
# Будуємо діаграму
# Отримуємо посилання на об'єкт типу matplotlib.axes.AxesSubplot,
# поточну діаграму
ax_01 = plt.axes()
# Задаємо вихідні дані для першої лінії діаграми
# (лінії ступеня розкладання), зовнішній вигляд лінії та #
маркера.
# Функція plot() повертає посилання на list,
#перший елемент, якого є об'єкт класу
# matplotlib.lines.Line2D
line_01 = ax_01.plot(X, Y_01, 'b-', label=leg_01)
# якщо потрібно надалі використовувати об'єкт класу
# matplotlib.lines.Line2D
# то застосовують ДВА варіанти
# 1) line_01 = ax_01.plot(X, Y_01, 'b-')
# отримання першого елемента списку
# 2) line_01 (без коми) = … використовувати line_01 [0]

# Задаємо інтервали значень по осях X та
# основної осі Y
ax_01.axis([-1, 13, -1.1, 1.1])

# Включаємо сітку по осі X та основної осі Y.
# Задаємо колір сітки
ax_01.grid(color = 'b',linewidth = 1)

# Задаємо підписи до осей X та основної осі Y
# та розмір шрифт
ax_01.set_xlabel('Час t sec', fontsize = 'x-large')
ax_01.set_ylabel(leg_01, color = 'b', fontsize = 'x-large')
# Задаємо заголовок діаграми
ax_01.set_title('Гармонічні сигнали '+ leg_01+ " та "+ \
                                          leg_02)
ax_01.legend( loc='upper right')

# Включаємо додаткову вісь Y
ax_02 = ax_01.twinx()

# Задаємо вихідні дані для другої лінії діаграми,
#  зовнішній вигляд лінії і маркера.
# Функція plot() повертає посилання на об'єкт класу
```

```
# matplotlib.lines.Line2D (див. вище)
line_02, = ax_02.plot(X, Y_02, 'r-', label=leg_02)
# Задаємо інтервали значень по осях X
# та додаткової осі Y
ax_02.axis([-1, 13, -12, 12])

# Задаємо підпис до додаткової осі Y
ax_02.set_ylabel(leg_02,color='r', fontsize = 'x-large')

# Задаємо вихідні дані для легенди та місце її розміщення
#ax_02.legend(loc='lower left')
ax_02.legend( loc='upper left')
# Включаємо сітку додаткової осі Y.
# Задаємо колір сітки
ax_02.grid(color = 'r')

# Зберігаємо побудовану діаграму у файл
# Задаємо ім'я файлу та його тип
#plt.savefig('trigo.png', format = 'png')
#візуалізуємо
plt.show()
```



Кілька графіків в одному та в різних графічних вікнах

Робота з matplotlib заснована на використанні графічних вікон та осей (осі дозволяють задати деяку графічну область). Усі побудови застосовуються до поточних осей. Це дозволяє зображувати декілька графіків в одному графічному вікні.

За замовчуванням створюється одне графічне вікно `figure(1)` та одна графічна область `subplot(111)` у цьому вікні.

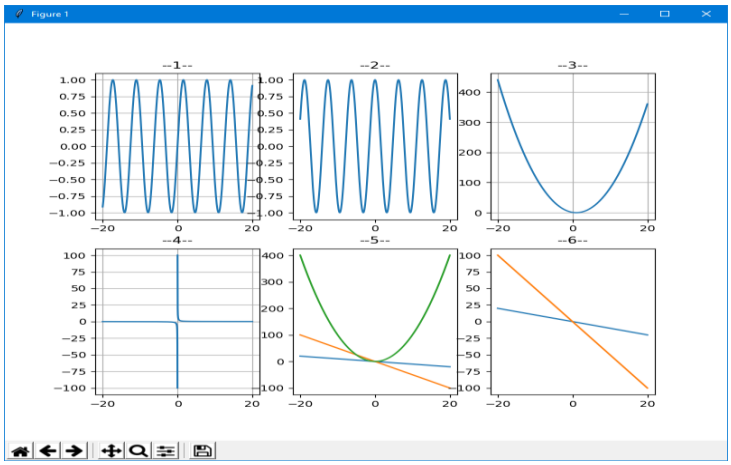
Команда `subplot` дозволяє розбити графічне вікно на кілька областей. Вона має три параметри: `nr`; `nc`; `np`:

- параметри nr та nc визначають кількість рядків та стовпців на які розбивається графічна область
- параметр np визначає номер поточної області (np набуває значення від 1 до nr*nc).

Якщо `nr*nc<10`, передавати параметри nr,nc,np можна без використання коми. Наприклад, допустимі форми `subplot(2,2,1)` та `subplot(221)`).

```
import math
import numpy as np
import matplotlib.pyplot as plt

# Імпортуємо пакет із допоміжними функціями
Import matplotlib.mlab as ml

# Малювати графік цієї функції
Def func (x):
  """
  sinc(x)
  """
  if x == 0:
      return 1.0
   return math.sin(x)/x


# Інтервал зміни змінної по осі X
xmin = -20.0
xmax = 20.0
# Крок між точками
dx = 0.01
# Створимо список координат по осі X на
# відрізку [-xmin; xmax], включаючи кінці
xlist = ml. frange(xmin, xmax, dx)
# Обчислимо значення функції у заданих точках
ylist = [func(x) for x in xlist]

# !!! Два рядки, три стовпці.
# !!! Поточний осередок - 1
plt.subplot (2, 3, 1)
plt.plot (xlist, np.sin(xlist) )   # ylist)
plt.grid(True)
plt.title ("--1--")

# !!! Два рядки, три стовпці.
# !!! Поточний осередок - 2
plt.subplot (2, 3, 2)
plt.plot (xlist, np.cos(xlist))
plt.title ("--2--")

# !!! Два рядки, три стовпці.
# !!! Поточний осередок - 3
plt.subplot (2, 3, 3)
plt.plot (xlist, (xlist-1)*(xlist-1))
plt.grid(True)
plt.title ("--3--")

# !!! Два рядки, три стовпці.
# !!! Поточний осередок - 4
```

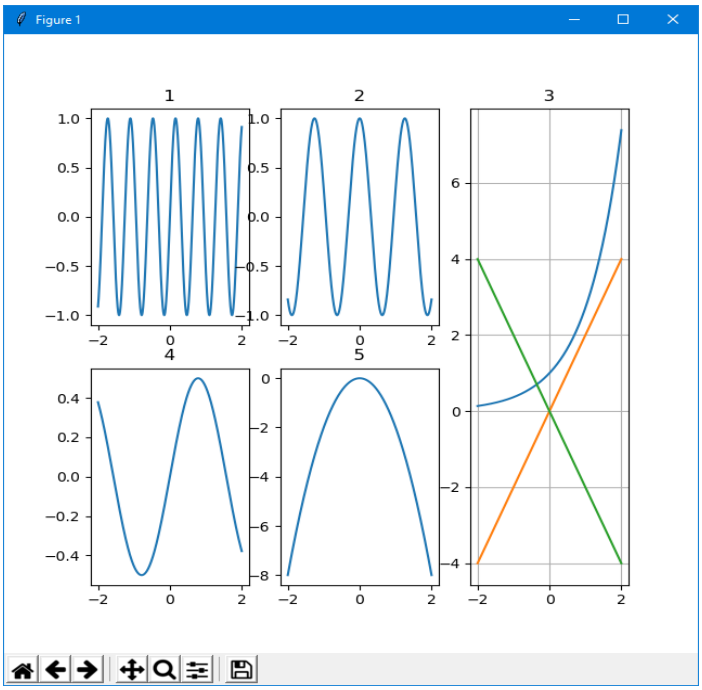
```
plt.subplot (2, 3, 4)
plt.plot (xlist, 1/xlist)
plt.grid(True)
plt.title ("--4--")

# !!! Два рядки, три стовпці.
# !!! Поточний осередок - 5
plt.subplot (2, 3, 5)
plt.plot (xlist, -xlist)
plt.plot (xlist, -5*xlist)
plt.plot (xlist, xlist * xlist)
plt.title ("--5--")

# !!! Два рядки, три стовпці.
# !!! Поточний осередок - 6
plt.subplot (2, 3, 6)
plt.plot (xlist, -xlist)
plt.plot (xlist, -5*xlist)
plt.title ("--6--")

# Покажемо вікно з намальованим графіком
plt.show()
```



Всі ці осередки суто умовні, тому для кожного виклику `subplot()` розбиття може бути своє, що дозволяє зробити, наприклад, наступне розташування графіків (зверніть увагу на нумерацію осередків):

```
import math
import numpy as np
import matplotlib.pyplot as plt
# Імпортуємо пакет із допоміжними функціями
import matplotlib.mlab as ml
# Інтервал зміни змінної по осі X
xmin = -2.0
xmax = 2.0
# Крок між точками
dx = 0.001
# Створимо список координат по осі X на відрізку [-xmin; xmax],
включаючи кінці
x = ml.frange (xmin, xmax, dx)
# !!! Два рядки, три стовпці.
# !!! Поточний осередок - 1
```

```
plt.subplot(2, 3, 1)
plt.plot (x, np.sin(10*x))
plt.title("1")
# !!! Два рядки, три стовпці.
# !!! Поточний осередок - 2
plt.subplot(2, 3, 2)
plt.plot(x, np.cos(5*x) )
plt.title("2")
# !!! Два рядки, три стовпці.
# !!! Поточний осередок - 4
plt.subplot (2, 3, 4)
plt.plot (x, np.sin(x)*np.cos(x))
plt.title ("4")
# !!! Два рядки, три стовпці.
# !!! Поточний осередок - 5
plt.subplot (2, 3, 5)
plt.plot (x, -2*x*x)
plt.title ("5")
# !!! Один рядок, три стовпці.
# !!! Поточний осередок - 3
plt.subplot (1, 3, 3)
plt.plot (x, np.exp(x))
plt.plot (x, 2*x)
plt.plot (x, -2*x)
plt.grid(True)
plt.title ("3")
# Покажемо вікно з намальованим графіком
plt.show()
```



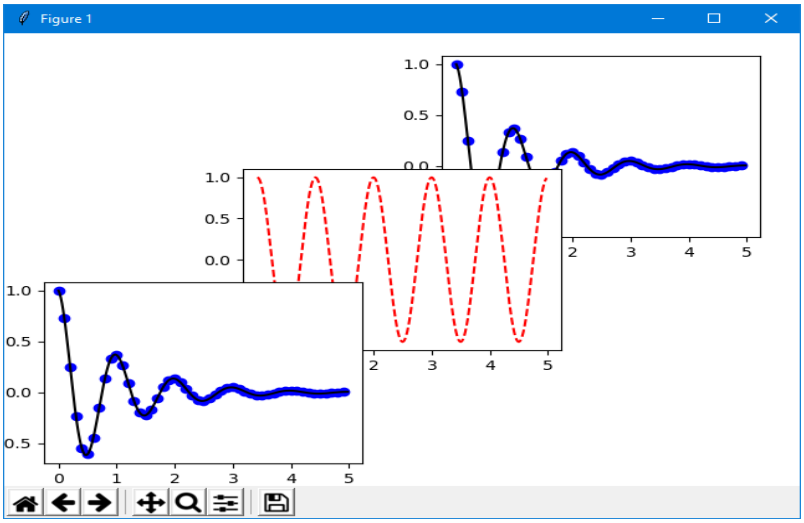
Розміщення координатних осей figure() та axes()

Графічні осі можна розміщувати вручну (тобто не на стандартній сітці), що дозволяє створювати нерегулярні або довільні макети. Для цього використовуйте функцію `plt.axes()`, яка дає змогу точно визначити положення кожної осі:

`axes([left, bottom, width, height]),`

де всі значення змінюються від 0 до 1. Виглядати це може так:

```
import math
import numpy as np
import matplotlib.pyplot as plt
# Імпортуємо пакет із допоміжними функціями
import matplotlib.mlab as ml
import numpy as np
import matplotlib.pyplot as plt
deff(t):
  return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
plt.figure(1)
plt.axes([0.55, 0.55, 0.4, 0.4])
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
plt.axes([0.3, 0.3, 0.4, 0.4])
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.axes([0.05, 0.05, 0.4, 0.4])
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
# Покажемо вікно з намальованим графіком
plt.show()
```

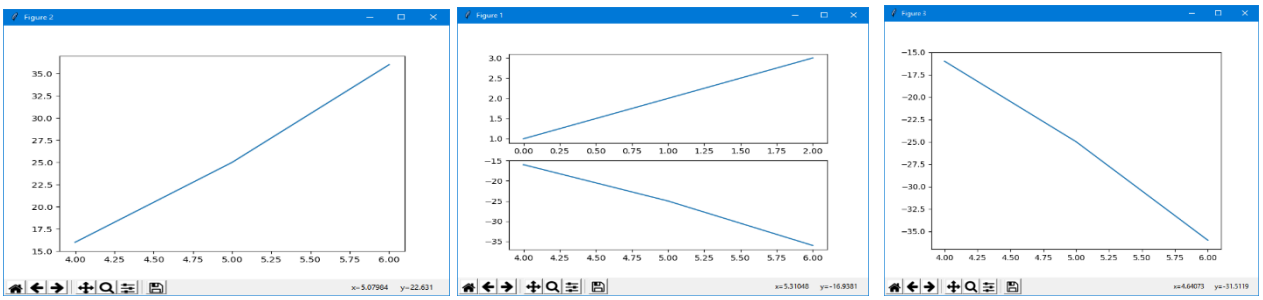


Є можливість створити кілька вікон, викликаючи `figure()` з номером вікна, що збільшується. Звичайно, кожне вікно може містити кілька осей та підвіконь.

Очистити активне вікно можна `clf()` , а активні осі за допомогою `cla()` .

```
import math
import numpy as np
import matplotlib.pyplot as plt
# Імпортуємо пакет із допоміжними функціями
import matplotlib.mlab as ml
import numpy as np
import matplotlib.pyplot as plt

plt.figure(1) # the first figure
plt.subplot(211) # 1
```

```
plt.plot([1,2,3])
plt.subplot(212) # second subplot in first figure
plt.plot([4,5,6],[-16,-25,-36])
plt.figure(2) # a second figure
plt.plot([4,5,6],[16,25,36]) # creates a subplot(111)
                                                by default
plt.figure(3)
plt.plot([4,5,6],[-16,-25,-36])
plt.show()
```



Встановити свій діапазон осей

```
plt.xlim(right=xmax)    #xmax is your value
plt.xlim(left=xmin)     #xmin is your value
plt.ylim(top=ymax)      #ymax is your value
plt.ylim(bottom=ymin)   #ymin is your value
```

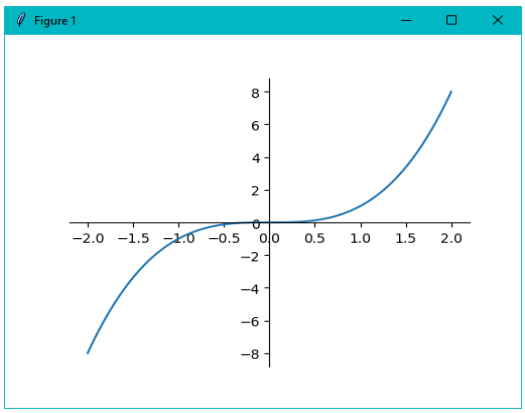
Або так:

```
plt.ylim(ymin, ymax)
plt.xlim(xmin, xmax)
```

Перенесення координатних осей до центру графіка

Якщо необхідно перенести координатні осі в центр рисунка, можна вчинити так:

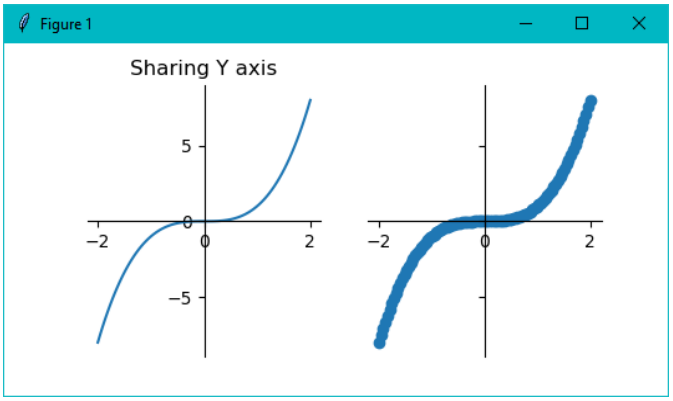
```
import matplotlib.pyplot as plt
import numpy as np
X = np.linspace(-2,2,100)
Y = X**3
plt.plot(X, Y)
ax = plt.gca()
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('center')
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
plt.show()
```



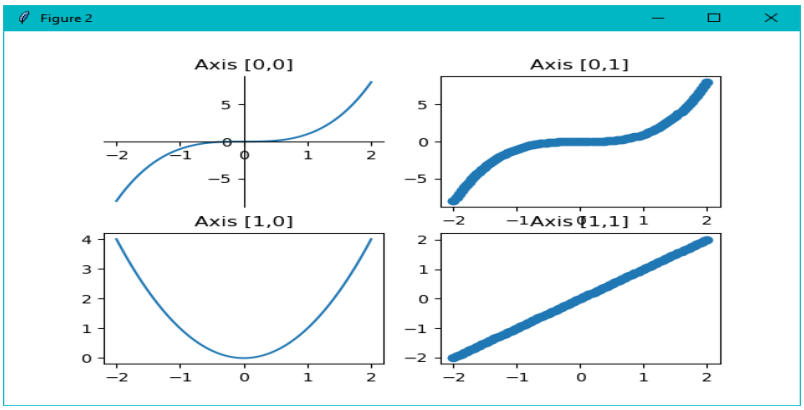
Приклад перенесення осей для subplot:

```
#Приклад отримання осей для subplots:
import matplotlib.pyplot as plt
import numpy as np

# Two subplots, unpack the axes array immediately
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(X, Y)
ax1.set_title('Sharing Y axis')
ax1.spines['left'].set_position('center')
ax1.spines['bottom'].set_position('center')
ax1.spines['top'].set_visible(False)
ax1.spines['right'].set_visible(False)
ax2.scatter(X, Y)
ax2.spines['left'].set_position('center')
ax2.spines['bottom'].set_position('center')
ax2.spines['top'].set_visible(False)
ax2.spines['right'].set_visible(False)
f, axarr = plt.subplots(2, 2) # Four axes,
                              # returned as a 2-d array
axarr[0, 0].plot(X, Y)
axarr[0, 0].spines['left'].set_position('center')
axarr[0, 0].spines['bottom'].set_position('center')
axarr[0, 0].spines['top'].set_visible(False)
axarr[0, 0].spines['right'].set_visible(False)
axarr[0, 0].set_title('Axis [0,0]')

axarr[0, 1].scatter(X, Y)

axarr[0, 1].set_title('Axis [0,1]')
axarr[1, 0].plot(X, X ** 2)
axarr[1, 0].set_title('Axis [1,0]')
axarr[1, 1].scatter(X, X ** 1)
axarr[1, 1].set_title('Axis [1,1]')


plt.show()
```

Перше вікно:



Друге вікно:

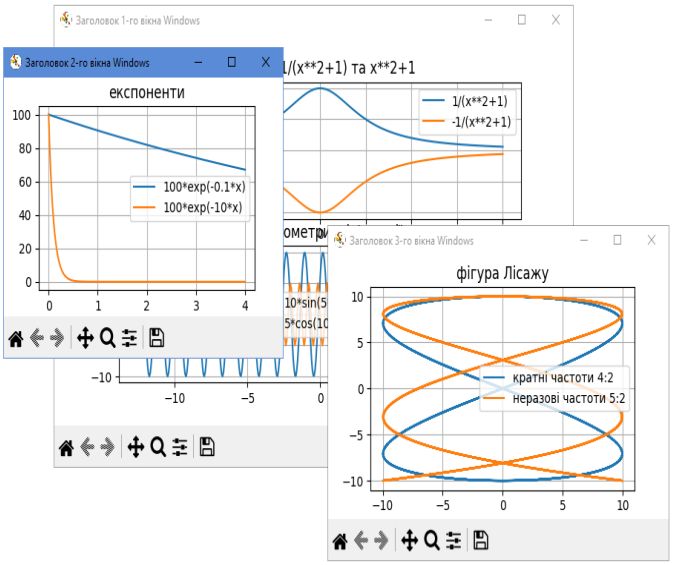


Напиши на вікнах діаграм і вікнах Windows, зміна розміру вікна

Наведемо приклад, аналогічний розглянутому вище. Для всіх вікон та діаграм задані титульні (тульбарні) написи:

```
import math
import numpy as np
import matplotlib.pyplot as plt
# Імпортуємо пакет із допоміжними функціями
import matplotlib as mpl
import numpy as np
import matplotlib.pyplot as plt
x1=np.linspace(-4,4,100)
y1=1/(x1**2+1)
y11=-1/(x1**2+1)
# Задамо розміри зображення діаграми
mpl.rcParams['figure.figsize'] = (8.0, 6.0)
plt.figure("Заголовок 1-го вікна Windows") # перше # вікно windows
plt.subplot(211) # перші графіки у цьому вікні
plt.plot(x1,y1,label='1/(x**2+1)')
plt.plot(x1,y11,label='-1/(x**2+1)')
plt.title("графіки 1/(x**2+1) та x**2+1" )

plt.grid(True)
plt.legend()
plt.subplot(212) # другі графіки у цьому вікні
x2=np.linspace(-4*np.pi,4*np.pi,400)
plt.plot(x2,10*np.sin(5*x2+0.2*np.pi),label='10*sin(5*x+0.2*pi)')
plt.plot(x2,5*np.cos(10*x2+0.5*np.pi),label="5*cos(10*x+0.5*pi)")
```

```
plt.title("тригонометричні функції")
plt.grid(True)
plt.legend(loc='best')

plt.figure("Заголовок 2-го вікна Windows") # a second figure
x3=np.linspace(0,4,400)
plt.plot(x3, 100*np.exp(-0.1*x3), label='100*exp(-0.1*x)')
plt.plot(x3, 100*np.exp(-10*x3), label='100*exp(-10*x)')

plt.title("експоненти")
plt.grid(True)
plt.legend(loc='best')
fig3=plt.figure("Заголовок 3-го вікна Windows")
plt.grid(True)
t=np.linspace(0,8*np.pi,500)
plt.plot(10*np.sin(4*t),10*np.cos(2*t),
                              label="кратні частоти 4:2")
plt.plot(10*np.sin(5*t),10*np.cos(2*t),
 label="неразові частоти 5:2")
plt.title("фігура Лісажу")
plt.grid(True)
plt.grid(True)
plt.legend(loc='best')
plt.show()
```



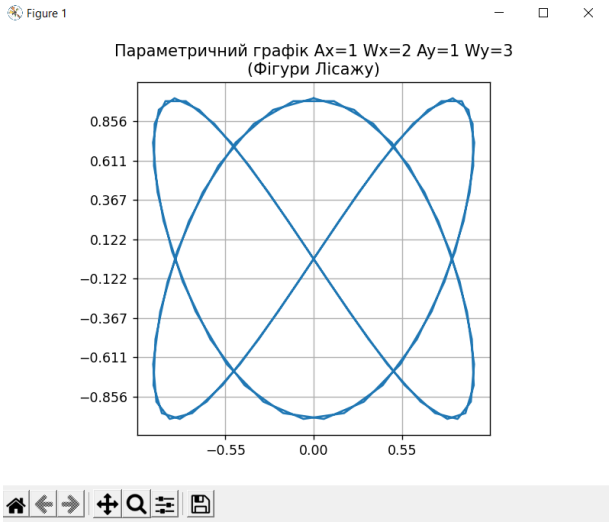
Параметричний графік

Масив x повинен бути монотонно зростаючим. Можна будувати будь-яку параметричну лінію x = x (t), y = y (t).

Роздільні можливості монітора по різних осях можуть бути різними. Тому щоб кола виглядали як кола, а не як еліпси (а квадрати як квадрати, а не як прямокутники), потрібно встановити параметр **aspect ratio** (зазвичай рівний 1).

Приклад. Побудуємо `x = Ax * cos (Wx * t),`
`y = Ay * sin (Wy * t).`

```
import matplotlib.pyplot as plt
import numpy as np
Ax=1; Ay=1; Wx = 2; Wy=3
# незалежна змінна
t=np.linspace(0,4*np.pi,100)
# Встановимо параметр - aspect (ratio)
axx=plt.subplot(111)
axx.set_aspect(1)

# так можна змінити крок сітки на графіку
plt.xticks([i for i in \
                      np.linspace(-1.1*Ax, 1.1*Ax, 5)])
plt.yticks([i for i in \
                    np.linspace(-1.1*Ay, 1.1*Ay, 10)]))

# Задаємо заголовок діаграми
plt.title("Параметричний графік" \
    " Ax={0} Wx={1} Ay={2} Wy={3}\n" \
    "(Фігури Лісажу)" \
.   format(Ax, Wx, Ay, Wy))

plt.plot(Ax*np.sin(Wx*t),Ay*np.cos(Wy*t))
plt.grid(True)
#візуалізуємо графіки
plt.show()
```

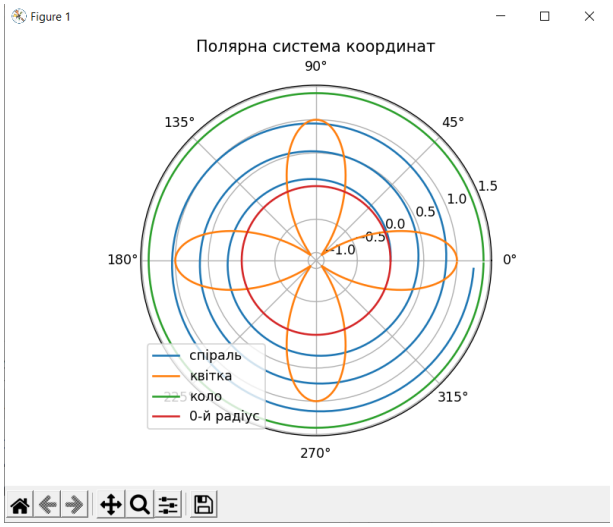


Полярні координати

Крім часто використовуваної декартової системи координат, досить широко застосовується і полярна система координат, зручна в різних радіальних задачах, координати точок в ній задається за допомогою радіус-вектора ρ, що йде з початку координат і кута θ. Кут може бути заданий у радіанах або градусах, matplotlib використовує градуси.

Чотири графіки в полярних координатах:

```
import matplotlib.pyplot as plt
import numpy as np

theta = np.arange(0., 2., 1./180.) * np.pi
#plt.title('Поляна система координат')
plt.polar(3*theta, theta/5, label="спіраль");
plt.polar(theta, 0+np.cos(4*theta), label="квітка");
plt.polar(theta, [1.4] * len (theta), label = "коло");
plt.polar(theta, 0*theta, label="0-й радіус");
plt.title("Полярна система координат")
plt.grid(True)
plt.legend(loc='lower left')
#візуалізуємо графіки
plt.show()
```



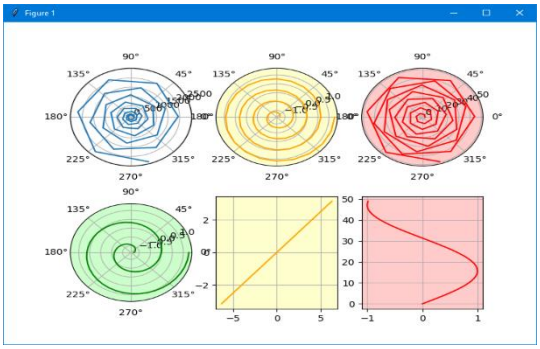
Для побудови в полярних координатах використовується функція `polar()`. В переліку аргументів першими йдуть кути, потім радіуси. У прикладі використовується той самий масив для кутів і радіус-векторів і малюються чотири різні криві.

Перша утворює спіраль, оскільки з кожним новим кутом змінюється і радіус, друга малює квітку відповідно до рівняння троянди, третя і четверта – кола через незмінність радіусу для всіх кутів (в даному випадку він дорівнює 1.4 і 0).

Ще один приклад:

```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure (figsize = (8., 6.))
x = np.arange(50)
y = x**2 + 2.5

lag = 0.05 * np.pi
r = np.arange(0.0, 5.0, 0.1)
# phi = r * np.pi
phi = np.arange (-2 * np.pi, 2 * np.pi + lag, lag)
```

```
r=phi*0.2
# x та y
ax1 = fig.add_subplot(231, projection='polar')
ax1.plot(x, y)
ax1.grid(True)

ax2 = fig.add_subplot(232, projection='polar',
                                  facecolor ='#FFFFCC')
ax2.plot(phi*2., r, 'orange')    # facecolor='#FFFFCC')
                                 #        ax is bg
ax2.grid(True)

ax3 = fig.add_subplot(233, projection='polar',
                                     facecolor='#FFCCCC')
ax3.plot(x, x, 'r')
ax3.grid(True)

ax4 = fig.add_subplot(234, projection='polar',
                                   facecolor='#CCFFCC')
ax4.plot(phi, 0.2*phi, 'g') #(-1.5*phi, r, 'g')
ax4.grid(True)

ax5 = fig.add_subplot(235, facecolor='#FFFFCC')
ax5.plot(phi, phi*0.5, 'orange')
ax5.grid(True)

ax6 = fig.add_subplot(236, facecolor='#FFCCCC')
ax6.plot(np.sin(0.1*x), x, 'r')
ax6.grid(True)
###plt.legend(loc='lower left')
plt.show()
```



`matplotlib` дозволяє малювати у полярної системі координат, а й у інших. За це відповідає параметр projection, який у разі рівності значенню 'polar' аналогічний значенню параметра polar=True.

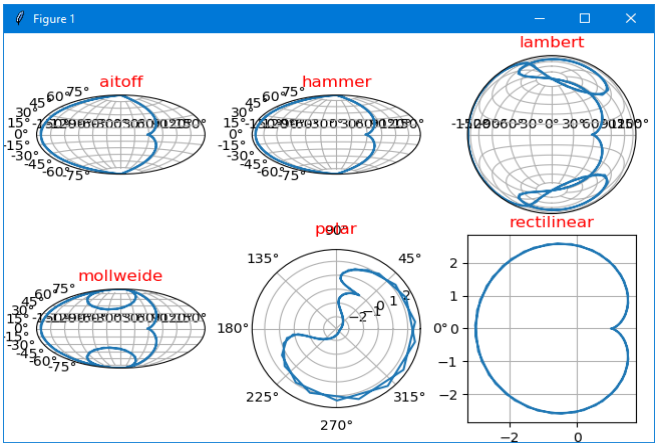
У matplotlib підтримуються такі проекції:

`'aitoff'`, `'hammer'`, 'lambert', `'mollweide'`, `'polar'`, `'rectilinear'`.

Найчастіше використовуються два останні варіанти, решта є екзотичними для рутинних завдань у науковій графіці.

```
import matplotlib.pyplot as plt
import numpy as np
```

```
a = 1.
x = np.arange(-2*np.pi, 2*np.pi, 0.2)
y = np.sin(x) * np.cos(x)
# Рівняння кардіоїди
xz = a * (2 * np.cos (x) - np.cos (2 * x))
yz = a*(2*np.sin(x) - np.sin(2*x))
label = ['aitoff', 'hammer', 'lambert', 'mollweide',
                                    'polar', 'rectilinear']
fig = plt.figure(figsize=(10,8))

for i in range(len(label)):
    ax = fig.add_subplot(231+i, projection=label[i])
    ax.plot(xz, yz)
    ax.set_title(label[i], color='r')
ax.grid(True)
plt.tight_layout()
plt.show()
```



Полярна система координат має спеціальні функції, які дозволяють налаштовувати зовнішній вигляд малюнка.

Для радіусу R:

- `ax.set_rlabel_position(phi)` – переміщає вісь ординат (радіус) по колу на кут `phi` (у градусах) від положення нуля;
- `ax.set_rmax(R)` – дозволяє обмежити область зміни радіуса R малюнку;
- `ax.set_rmin(R)` – дозволяє обмежити область зміни радіуса R малюнку;
- `ax.set_rlim()` – дозволяє обмежити область зміни радіуса R малюнку;
- `ax.set_rscale()` – дозволяє зробити шкалу радіусів логарифмічної;
- `ax.set_rgrid()` – дозволяє налаштувати для осі радіусу допоміжну сітку, положення поділів, формат підписів до них тощо.

Для кута phi (в matplotlib він називається theta):

- `ax.set_theta_zero_location(loc)` – Переміщає положення нуля на певне положення. Loc приймає значення `'N', 'NW', 'W', 'SW', 'S',`

'SE', 'E' або 'NE'. За умовчанням положення нуля знаходиться у положенні "схід" або "3 години";

- ax.set_theta_offset(phi) – Переміщає положення нуля на кут phi (у радіанах). За умовчанням положення нуля знаходиться у положенні "схід" або "3 години";
- ax.set_theta_direction(loc) – Визначає напрямок обходу. Loc може бути -1 або за годинниковою стрілкою і 1 або проти годинникової стрілки;
- ax.set_thetagrids() – дозволяє налаштувати для осі кутів допоміжну сітку, положення поділів, формат підписів до них тощо.

Графік розсіювання

Такий тип графіків дозволяє зображати одночасно дві множини даних, які не утворюють кривої, а саме двовимірне безліч точок. Кожна точка має дві координати. Графік розсіювання часто використовується визначення зв'язку між двома величинами і дозволяє визначити більш точні межі вимірів.

У модулі matplotlib.pyplot є своя функція, для графіка розсіювання (scatter plot) це функція scatter().

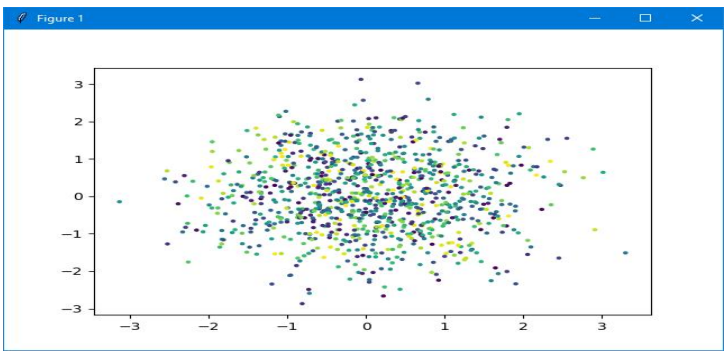
Вона приймає дві послідовності та зображує їх на площині у вигляді маркерів, за умовчанням вони круглі та сині. Але природно, з ними можна попрацювати за допомогою keywords тієї ж функції:

s задає розмір маркерів і може бути одним числом для всіх, так і представляти масив значень

c задає колір маркерів, або один для всіх, або безліч

marker визначає тип маркеру.

```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.randn(1000); y = np.random.randn(1000)
size = 5
colors = np.random.rand(1000)
plt.scatter(x, y, s=size, c=colors)
plt.show()
```



Налаштування в стилі LATEX

Ось приклад налаштування майже всього, що можна налаштувати.

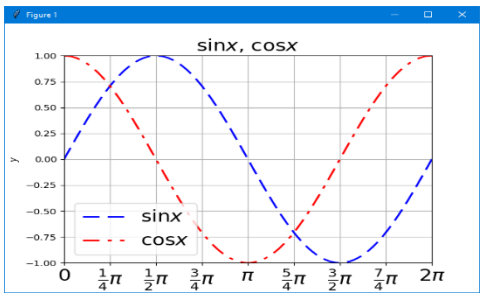
Можна задати послідовність засічок на осі x (і y) та підписи до них (у них, як і в інших текстах, можна використовувати LATEX-івські позначення).

Задати підписи осей x та y та заголовок графіка.

У всіх текстових елементах можна встановити розмір шрифту. Можна задати товщину ліній і штрихи (так, на графіку косинуса малюється штрих довжини 8, потім ділянка довжини 4 не малюється, потім ділянка довжини 2 малюється, потім ділянка довжини 4 знову не малюється, і так за циклом; оскільки товщина лінії дорівнює 2, ці короткі штрихи довжини 2 фактично виглядають як крапки).

Можна поставити підписи до кривих (legend); де розмістити ці підписи також можна регулювати

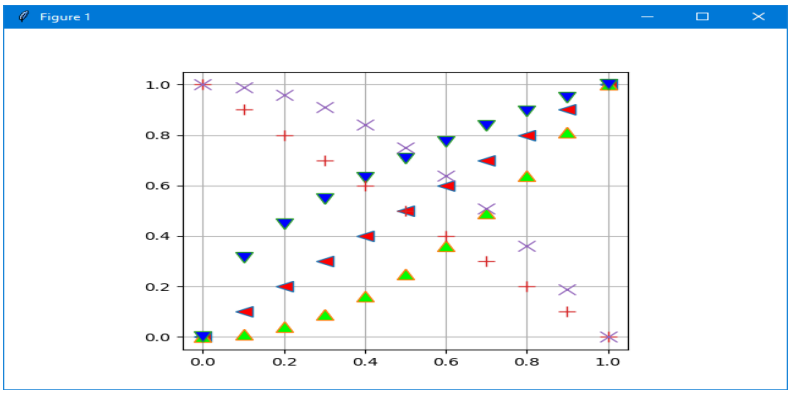
```
import matplotlib.pyplot as plt
import numpy as np
plt.axis([0,2*np.pi,-1,1])

plt.xticks(np.linspace(0,2*np.pi,9),
 ('0',r'$\frac{1}{4}\pi$',r'$\frac{1}{2}\pi$',
 r'$\frac{3}{4}\pi$',r'$\pi$',r'$\frac{5}{4}\pi$',
 r'$\frac{3}{2}\pi$',r'$\frac{7}{4}\pi$',r'$2\pi$'),
 fontsize=20)

plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'$\sin x$, $\cos x$',fontsize=20)

x=np.linspace(0,2*np.pi,100)
plt.plot(x,np.sin(x),linewidth=2,
                        color='b',
                           dashes=[8,4],
                                  label=r'$\sin x$')

plt.plot(x,np.cos(x),linewidth=2,
                     color='r',dashes=[8,4,2,4],
                                       label=r'$\cos x$')

plt.legend(fontsize=20)
plt.grid(True)

#візуалізуємо графіки
plt.show()
```



Модифіковані маркери

Якщо `linestyle=''` то точки не з'єднуються лініями. Самі крапки малюються маркерами різних типів. Тип визначається рядком з одного символу, який чимось нагадує потрібний маркер. На додаток до стандартних маркерів, можна визначити саморобні.

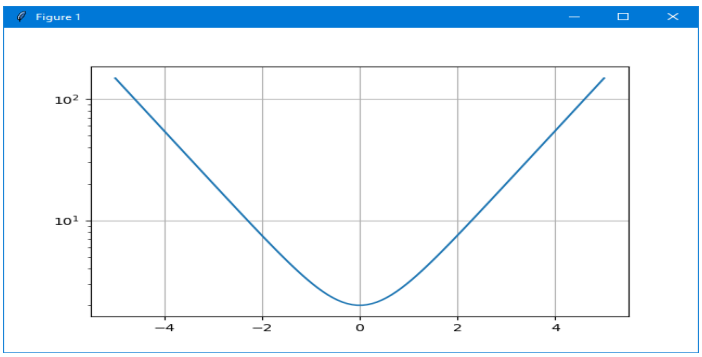
```
impor matplotlib.pyplot as plt
import numpy as np
x=np.linspace(0,1,11)
# Встановимо параметр - aspect (ratio)
axx=plt.subplot(111)
axx.set_aspect(1)
plt.axis([-0.05,1.05,-0.05,1.05])
plt.plot(x,x,linestyle='',marker='<',markersize=10,
                              markerfacecolor='#FF0000')
plt.plot(x,x**2,linestyle='',marker='^',markersize=10,
                               markerfacecolor='#00FF00')
plt.plot(x,x**(1/2),linestyle='',marker='v',
                                     markersize=10,
                               markerfacecolor='#0000FF')
plt.plot(x,1-x,linestyle='',marker='+',markersize=10,
                               markerfacecolor='#0F0F00')
plt.plot(x,1-x**2,linestyle='',marker='x',markersize=10,
                               markerfacecolor='#0F000F')
plt.grid(True)
візуалізуємо графіки
plt.show()
```



Логарифмічний масштаб

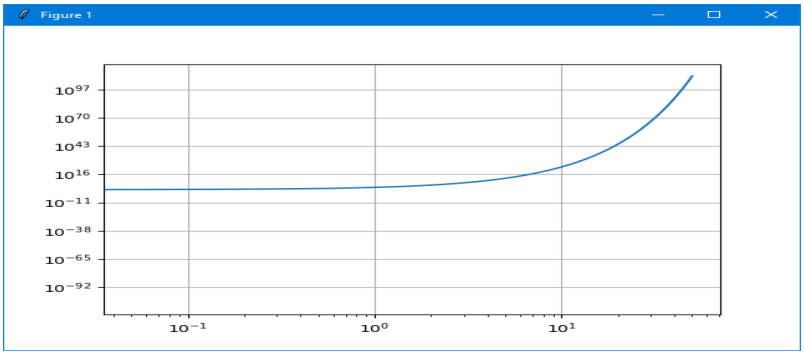
Якщо у змінюється на багато порядків, то зручно використовувати логарифмічний масштаб по y:

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(-5,5,100)
plt.yscale('log')
plt.plot(x,np.exp(x)+np.exp(-x))
plt.grid(True)
візуалізуємо графіки
plt.show()
```



Можна встановити логарифмічний масштаб за обома осями.

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(-50,50,1000)
plt.xscale('log')
plt.yscale('log')
plt.plot(x,np.exp(5*x+3))
plt.grid(True)
візуалізуємо графіки
plt.show()
```



Експериментальні дані

Припустимо, є теоретична крива (наприклад, резонанс без фону).

```
xt=np.linspace(-4,4,101)
yt=1/(xt**2+1)
```

Оскільки реальних експериментальних даних немає, ми їх згенеруємо. Нехай вони узгоджуються з теорією і всі статистичні помилки дорівнюють 0.1.

```
xe=np.linspace(-3,3,21)
yerr=0.1*np.ones(21)
ye=1/(xe**2+1)+yerr*np.random.normal(size=21)
```

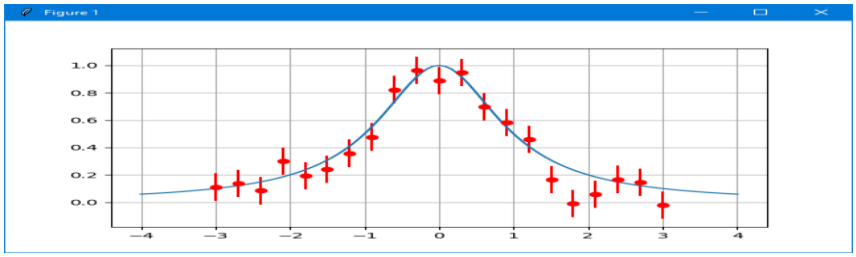
Експериментальні точки з вусами і теоретична крива однією графіку.

```
plt.plot(xt,yt)
plt.errorbar(xe,ye,fmt='ro',yerr=yerr)
```

Весь скрипт:

```
import math
import numpy as np
import matplotlib.pyplot as plt
```

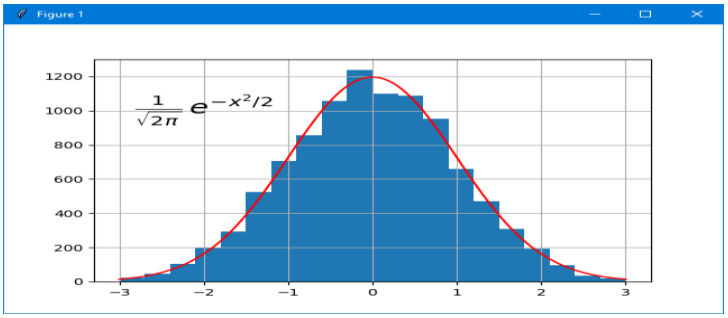
```
xt=np.linspace(-4,4,101)
yt=1/(xt**2+1)
xe=np.linspace(-3,3,21)
yerr=0.1*np.ones(21)
ye=1/(xe**2+1)+yerr*np.random.normal(size=21)
plt.plot(xt,yt)
plt.errorbar(xe,ye,fmt='ro',yerr=yerr)
plt.grid(True)
візуалізуємо графіки
plt.show()
```



Гістограма

Згенеруємо N випадкових чисел з нормальним (гаусовим) розподілом (середнє 0, середньоквадратичне відхилення 1), і розкидаємо їх по 20 бін від -3 до 3 (точки за межами цього інтервалу відкидаються). Для порівняння, разом із гістограмою намалюємо Гауссову криву у тому самому масштабі. І навіть напишемо формулу Гауса.

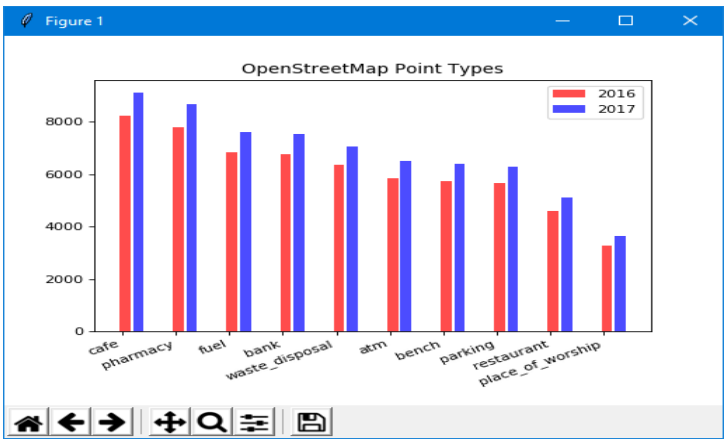
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as ml
N=10000
r=np.random.normal(size=N)
n, bins, patches = plt.hist (r, range = (-3,3),
                                                bins = 20)
x=np.linspace(-3,3,100)
plt.plot(x,N/np.sqrt(2*np.pi)*0.3* np.exp(-0.5*x**2),'r')

plt.text(-2,1000,r'$\frac{1}{\sqrt{2\pi}} \,e^{-x^2/2}$',
                        fontsize=20,horizontalalignment='центр',
                                        verticalalignment='center')
plt.grid(True)
#візуалізуємо графіки
plt.show()
```



Розглянемо кілька прикладів (https://eax.me/python-matplotlib/) побудови спеціальних діаграм.

Як приклад побудуємо діаграму, що відображає, скільки точок на карті якого типу (заправка, кафе тощо) ставляться. Щоб було трохи цікавіше, вдамо, що торік точок кожного виду було на 10% менше, і спробуємо відобразити цю зміну:

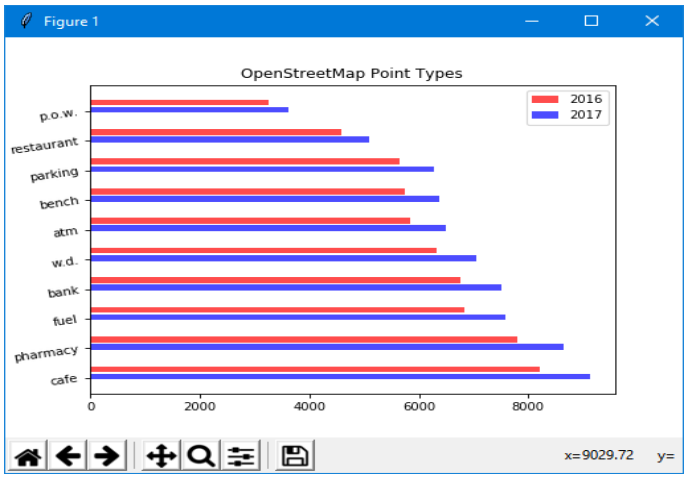
```
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.dates як mdates
import datetime as dt
import csv
data_names = ['cafe', 'pharmacy', 'fuel', 'bank',
              'waste_disposal', 'atm', 'bench',
          'parking', 'restaurant', 'place_of_worship']
data_values = [9124, 8652, 7592, 7515, 7041, 6487, 6374,
                                        6277,  5092, 3629]
dpi = 80
fig = plt.figure(dpi=dpi, figsize=(512/dpi, 384/dpi) )
mpl.rcParams.update({'font.size': 10})
plt.title('OpenStreetMap Point Types')
#ax = plt.axes()
#ax.yaxis.grid(True, zorder = 1)
xs = range(len(data_names))
plt.bar([x + 0.05 for x in xs],
        [d * 0.9 for d in  data_values],
        width = 0.2, color = 'red', alpha = 0.7,
        label = '2016',  zorder = 2)
plt.bar([x + 0.3 for x in xs], data_values,
         width = 0.2, color = 'blue', alpha = 0.7,
         label = '2017',  zorder = 2)
plt.xticks(xs, data_names)
fig.autofmt_xdate(rotation = 25)
plt.legend(loc='upper right')
#fig.savefig('bars.png')
plt.show()
```



Ті ж дані можна відобразити, розташувавши стовпчики горизонтально:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.dates як mdates
import datetime as dt
```

```
import csv

data_names = ['cafe', 'farmaceu', 'fuel', 'bank', 'wd',
         'atm', 'bench', 'parking', 'restaurant', 'pow']
data_values  [9124, 8652, 7592, 7515, 7041, 6487, 6374,
                                      6277, 5092, 3629]

dpi = 80
fig = plt.figure(dpi = dpi, figsize =
                                   (512/dpi, 384/dpi))
mpl.rcParams.update({'font.size': 9})
plt.title('OpenStreetMap Point Types')
#ax = plt.axes()
#ax.xaxis.grid(True, zorder = 1)
xs = range(len(data_names))
plt.barh([x + 0.3 for x in xs],
         [ d * 0.9 for d in data_values],
           height = 0.2, color = 'red',
              alpha = 0.7, label = '2016', zorder = 2)
plt.barh([x + 0.05 for x in xs], data_values,
            height = 0.2, color = 'blue',
           alpha = 0.7, label = '2017',  zorder = 2)
plt.yticks(xs, data_names, rotation = 10)

plt.legend(loc='upper right')
plt.show()
```



Кругова діаграма

Наприклад візуалізуємо розподіл кафе по різних містах:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import datetime as dt
import csv
data_names = [
          "Київ",  "Львів", "Полтава",
          "Вінниця", "Миколаїв",  "Одеса",  "Харків",
          "Рівне",  "Дніпро",  "Херсон" ]
```

```
data_values= [1076, 979, 222, 189, 137, 134, 124, 124,
                                                  91, 79]
dpi = 80
fig = plt.figure(dpi=dpi, figsize=(512/dpi, 384/dpi) )
mpl.rcParams.update({'font.size': 9})
plt.title('Розподіл кафе по містах України')
xs = range(len(data_names))
plt.pie(
  data_values, autopct='%.1f', radius = 1.1,
  explode = [0.15] +
     [0 for _ in range(len(data_names) - 1)] )
plt.legend(
   bbox_to_anchor = (-0.16, 0.45, 0.25, 0.25),
      loc = 'lower left', labels = data_names )
plt.show()
```



Текст та написи

Текст та додаткові написи (анотації) розміщуються на діаграмі:

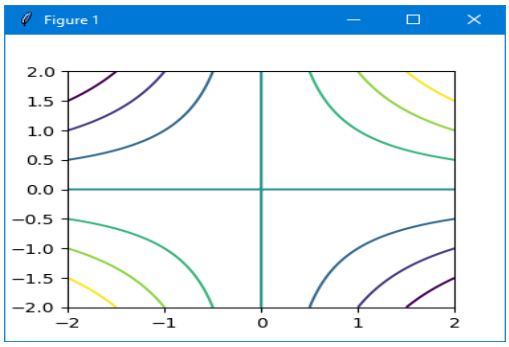
```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
ax = plt.subplot(111)

t = np.arange(0.0, 5.0, 0.01)
s = (1-(t-2)*(t-2)) * np.cos(2*np.pi*t)
line, = plt.plot (t, s, lw = 2)
plt.text(1,-1.5,'це текстовий напис', size=20, color='r')
plt.grid(True)
plt.annotate('локальний максимум', xy=(2, 1), xytext=(3, 1.5),
arrowprops= dict(facecolor='black', shrink=0.05), )
plt.ylim(-2,2)
plt.show()
```



Контурні графіки

Нехай потрібно вивчити поверхню **z = xy**. Побудуємо її «горизонталі»:

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
x=np.linspace(-1,1,50)
y=x
z = np.outer (x, y)
plt.contour(x,y,z)
plt.axes().set_aspect(1)
plt.show()
```



Функція `np.outer(x, y)` обчислює “зовнішнє добуток” x на y. Створюється матриця, де кожен елемент є добутком відповідного елемента з x на відповідний елемент з y.

Якщо `x` – вектор-рядок, а `y` – вектор-стовпець, результатом є матриця, яку можна представити як добуток цієї матриці-рядка на цю матрицю-стовпець.

`np.outer(x, y)` приймає два одновимірні масиви (вектори) x і y і повертає двовимірний масив (матрицю).

Розмір підсумкової матриці дорівнюватиме [розмір x, розмір y]

Приклад використання:

```
import numpy as np
# Створюємо два одномірні масиви (вектори)
x = np.array([1, 2, 3])
y = np.array([4, 5])
# Обчислюємо зовнішній твір
result = np.outer(x, y)
print(result)
```

Результат виконання коду

```
[[ 4,  5]
 [ 8, 10]
 [12  15]]
```

Пояснення `result = np.outer(x, y)`:

Перший елемент `x=1` множиться на кожен елемент `y [4, 5]`, даючи перший рядок результату: `[1*4, 1*5] = [4, 5];`

Другий елемент `x=2` множиться на кожен елемент `y [4, 5]`, даючи другий рядок результату: `[2*4, 2*5] = [8, 10];`

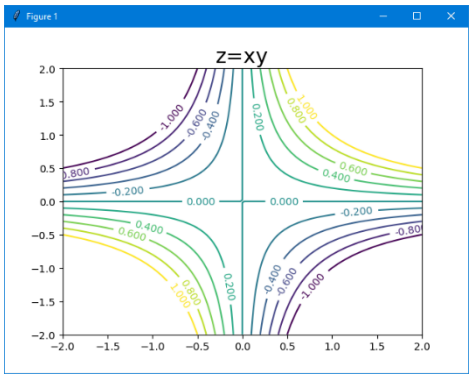
Третій елемент `x=3` множиться на кожен елемент `y [4, 5]`, даючи третій рядок результату: `[3*4, 3*5] = [12, 15].`

Ще приклад:

```
>>> x=np.array([1,2,3])
>>> y=np.array([10,20,30])
>>> x*y
array([10, 40, 90])
>>> np.outer(x,y)
array([[10, 20, 30],
       [20, 40, 60],
       [30, 60, 90]])
```

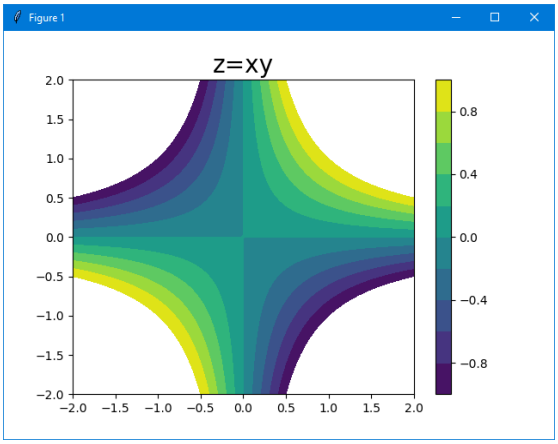
Горизонталі можна розфарбувати і підписати, а також збільшити їх кількість:

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
x=np.linspace(-2,2,100)
y=x
z = np.outer (x, y)
plt.title('z=xy',fontsize=20)
curves=plt.contour(x,y,z,np.linspace(-1,1,11))
plt.clabel(curves)
plt.show()
```



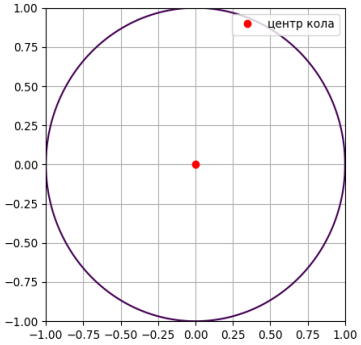
Висоту (значення Z) можна встановити кольором, як на фізичних географічних картах. `colorbar` показує відповідність кольорів та значень z.

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
x=np.linspace(-2,2,100); y=x
z = np.outer (x, y)
plt.title('z=xy',fontsize=20)
plt.contourf(x,y,z,np.linspace(-1,1,11))
plt.colorbar()
#plt.axes().set_aspect(1)
plt.show()
```



Можна «малювати» графіки функцій на площині, що задані «неявно»: F(x, y)=0:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-1.0, 1.0, 100)
y = np.linspace(-1.0, 1.0, 100)
X, Y = np.meshgrid(x,y)
F = X**2 + Y**2 - 1   #0.6
plt.contour(X,Y,F,[0])
plt.plot([0],[0],'ro', label="центр кола")
plt.gca().set_aspect('equal') #, щоб малюнок
                              #  виглядав кругом

# Включаємо сітку
plt.grid(True)
plt.legend(loc='best')
plt.show()
```

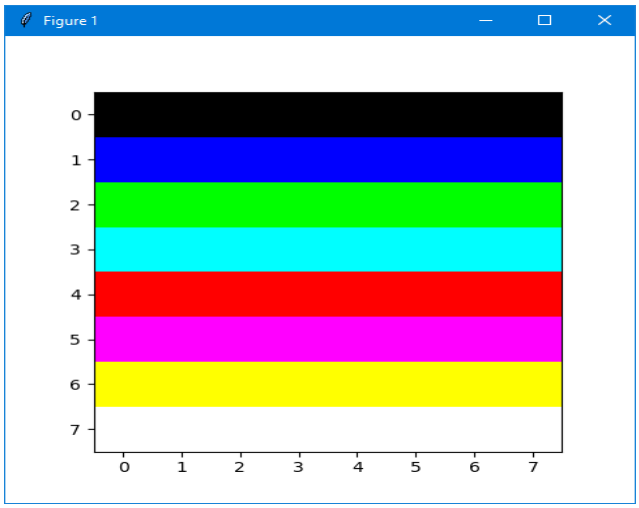


Images (піксельні картинки)

Картинка задається масивом `z`: значення `z[i,j]` – це колір пікселя i, j, масив із 3 елементів `rgb`. Де `rgb` числа від 0 до 1.0 (`float`) або від 0 до 255 (`int`) є інтенсивності кольорів – червоного, зеленого та синього.

Для наочності у прикладі формуються «натуральні» кольори до списку `col`:

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
n=8
u=np.linspace(0,1,n)
x,y=np.meshgrid(u,u)
z=np.zeros((n,n,3))
col=[]
for r in range(0,2):
for g in range(0,2):
for b in range(0,2):
   print(r, g, b)
   col.append([r, g, b])
#print (col)
for i in range(n):
for j in range(n): # RGB
   z[i,j]=col[i]
plt.imshow(z)
plt.show()
```



Ще приклад.

Функція `plt.imshow()` з бібліотеки `Matplotlib` відображає двовимірні дані, наприклад, масив NumPy, як зображення або теплову карту, де кожен елемент даних представлений кольором відповідно до карти кольорів. На вхід функції подається двовимірний масив даних z, а виведення використовується `plt.show()`.

Приклад використання:

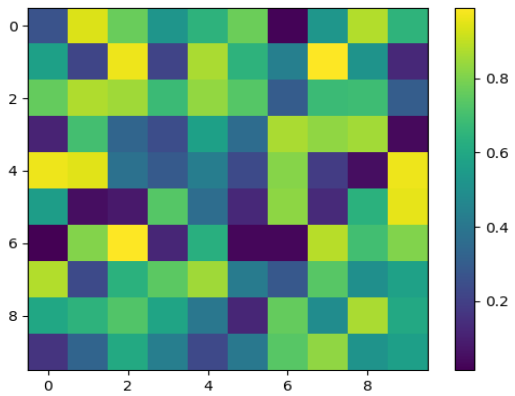
```
import matplotlib.pyplot as plt
import numpy as np
```

```
# Створюємо двовимірний масив даних (імітація зображення)
# Наприклад, масив 10x10 з різними значеннями
z = np.random.rand(10, 10)

# Відображаємо масив як зображення
plt.imshow(z, cmap='viridis') # 'viridis' -
                                приклад колірної карти

# Додаємо колірну шкалу для розуміння значень
plt.colorbar()

# Відображаємо отриманий графік
plt.show()
```

Пояснення коду:

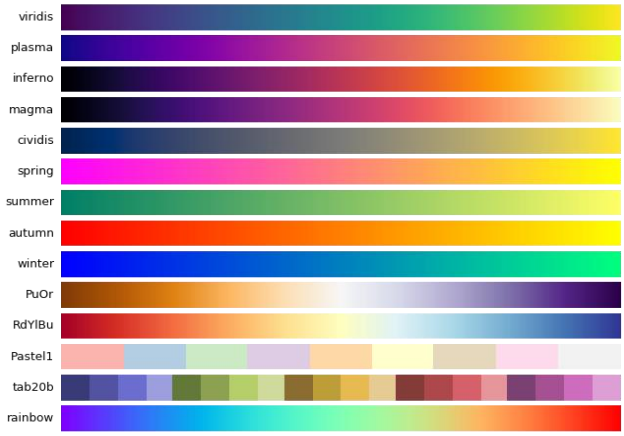
1. `import matplotlib.pyplot as plt` та `import numpy as np`: імпортуємо необхідні бібліотеки. plt для функцій побудови графіків, а np створення масиву даних.
2. `z = np.random.rand(10, 10)`: створюємо двовимірний масив (матрицю) розміром 10x10, заповнений випадковими числами від 0 до 1. Це будуть наші дані для відображення
3. `plt.imshow(z, cmap = 'viridis')`:
   - `plt.imshow(z)`: основна команда, яка бере масив z і перетворює його на зображення.
   - `cmap='viridis'`: цей параметр задає картку кольорів. Колірна карта визначає, який колір буде призначено кожному значенню у вашому масиві. У цьому випадку використовується колірна карта `'viridis'`, яка переводить скалярні дані в кольори.
4. `plt.colorbar()`: додає колірну шкалу (легенду) праворуч від зображення, яка показує відповідність між кольорами та числовими значеннями даних.
5. `plt.show()`: відображає отримане зображення на екрані.

Таким чином, `plt.imshow(z)` перетворює числові дані у візуальне уявлення, де кольори відповідають значенням, що особливо корисно для аналізу теплових карт, зображень та інших двовимірних даних



Кольорова карта є підготовленим набором кольорів, який добре підходить для візуалізації того чи іншого набору даних. Детальний посібник з кольорових карт ви можете знайти на офіційному сайті `Matplotlib`.

Також зазначимо, що такі карти можна створювати самостійно, якщо серед існуючих немає відповідного рішення. Нижче наведено приклади деяких колірних схем з бібліотеки `Matplotlib`.



Тривимірний графік

Усі класи для роботи з тривимірними графіками знаходяться у пакеті `mpl_toolkits.mplot3d`, з якого потрібно їх імпортувати.

Для того щоб намалювати тривимірний графік, в першу чергу треба створити тривимірні осі.

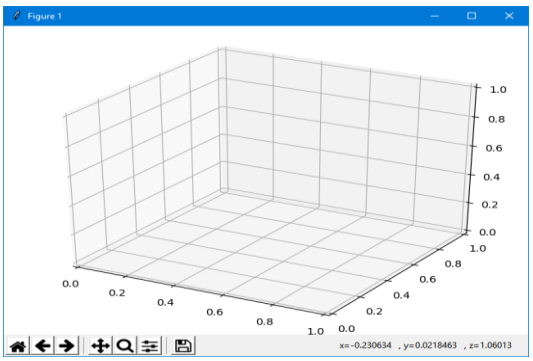
Щоб створити їх, потрібно створити екземпляр класу `mpl_toolkits.mplot3d.Axes3D`. Його конструктор чекає, як мінімум, один параметр – екземпляр класу `matplotlib.figure.Figure`. Цей об'єкт створюється `pylab.figure()`.

Конструктор класу `matplotlib.figure.Figure` має ще й інші необов'язкові параметри.

Намалюємо порожні осі:

```
import numpy as np
import pylab
from mpl_toolkits.mplot3d import Axes3D

fig = pylab.figure()
Axes3D(fig)
pylab.show()
```

В результаті побачимо наступне вікно:



Отримані осі можна обертати мишкою.

Тривимірна лінія

Лінія в просторі задається параметрично: x = x (t), y = y (t), z = z (t). Наприклад так:

```
t=np.linspace(0, 4*np.pi,1000)
x=np.cos(2*t)
y=np.sin(2*t)
z = t / (4 * np.pi)
```

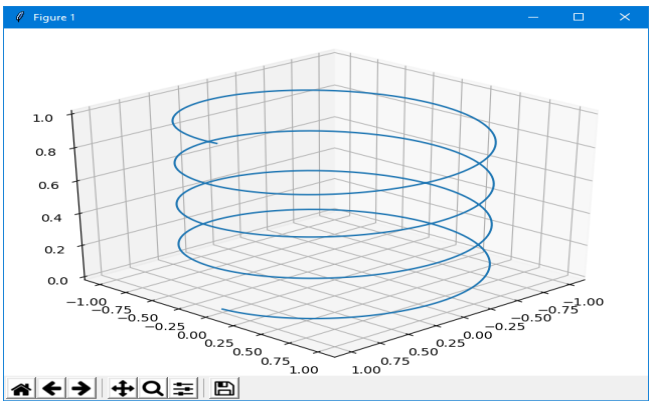
Для побудови та візуалізації використовується об'єкт класу `Axes3D` з пакету `mpl_toolkits.mplot3d`:

```
import pylab
import mpl_toolkits.mplot3d as A3D
```

`figure()` – це поточний малюнок, створюємо в ньому об'єкт ax, потім використовуємо його методи для візуалізації

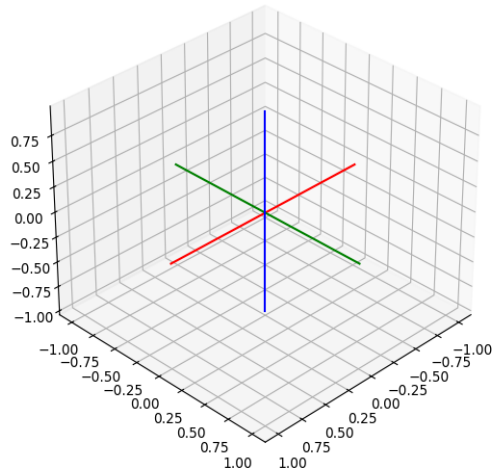
```
import pylab
import numpy as np
impor tmatplotlib as mpl
import matplotlib.pyplot as plt
##from mpl_toolkits.mplot3d import Axes3D // old version
import mpl_toolkits.mplot3d as A3D
t=np.linspace(0, 4*np.pi,1000)
x=np.cos(2*t)
y=np.sin(2*t)
z = t / (4 * np.pi)

#Тут потрібен об'єкт класу Axes3D з пакету
#                       mpl_toolkits.mplot3d.
# figure() - це поточний малюнок,
#  створюємо в ньому об'єкт axes, потім використовуємо методи.
#fig=pylab.figure()  // old version
#ax=A3D.Axes3D(fig)  // old version
fig = pylab.figure()
axes = A3D.Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(axes)
ax.elev,ax.azim=30,45 # задати, з якого боку дивимося.

ax.plot3D(x,y,z)
```

```
plt.show()
```



Побудуємо три прямі по осях координат:

```
import pylab
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d as A3D

fig = pylab.figure()
axes = A3D.Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(axes)
axes.elev,axes.azim=30,45 # specify which side we are looking from

t = np.arange(-1, 1, 0.01)
#           X    Y    Z
axes.plot3D(  t, 0*t, 0*t, color='r' )
axes.plot3D(0*t,   t, 0*t, color='g' )
axes.plot3D(0*t, 0*t,   t, color='b' )

plt.show()
```



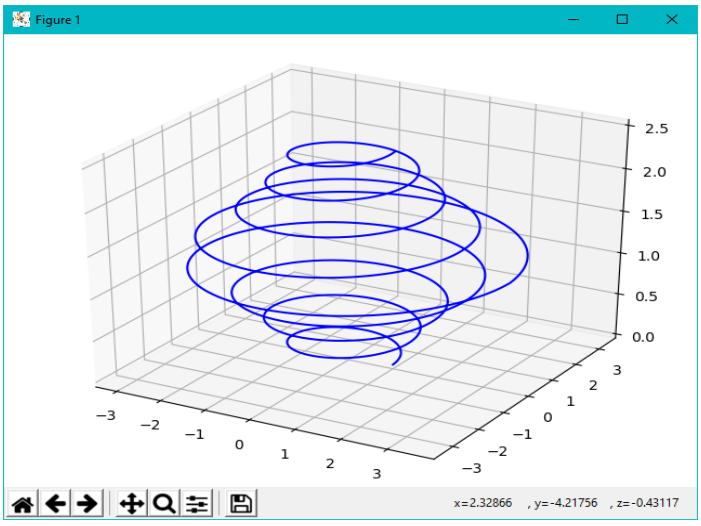
Побудуємо лінію, задану рівняннями:

$$x(t) = \cos(2 \cdot t) \cdot e^{\frac{a \cdot t}{10}}, \quad y(t) = \sin(2 \cdot t) \cdot e^{\frac{a \cdot t}{10}}, \quad z(t) = \frac{t}{10}, \quad t \in [0{,}8 \cdot \pi]$$

$$a = \{-1, \text{при} t \in$$

```
import numpy as np
```

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d as A3D
tt1 = 0; tt2 = 4 * np.pi; tt3 = 8 * np.pi; n1 = 500; n2=500
#-------
t1 = np.linspace (tt1, tt2, n1)
x1=np.cos(2*t1) *np.exp(0.1*t1)
y1=np.sin(2*t1) *np.exp(0.1*t1)
z1 = 0.1 * t1
#------
ttt=t1[500-1]
t2 = np.linspace (tt2, tt3, n2)
x2=np.cos(2*(t2)) *np.exp(-0.1*(t2-ttt)+0.1*ttt) #+ x1[500-2]
y2=np.sin(2*(t2)) *np.exp(-0.1*(t2-ttt)+0.1*ttt) #+ y1[500-2]
z2 = 0.1 * t2
#-----
#print("x=", x1[500-1], x2[0], t1[500-1], t2[0])
#print("y=", y1[500-1], y2[0], t1[500-1], t2[0])
#-----
x=np.concatenate((x1[:-1],x2))
y=np.concatenate((y1[:-1],y2))
z=np.concatenate((z1[:-1],z2))
#----
fig=plt.figure()
ax = A3D.Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(ax)
ax.plot3D(x,y,z,'b-')
plt.show()
```



Поверхні

Усі поверхні задаються параметрично: x = x (u, v), y = y (u, v), z = z (u, v).

Якщо необхідно задати поверхню «явно» z=z(x,y), то зручно створити масиви x=u і y=v функцією `meshgrid`.

Для всіх прикладів будемо використовувати наступну функцію від двох координат.

$$f(x,y)=\frac{\sin(x)\sin(y)}{xy}$$

Спочатку потрібно підготувати дані для малювання. Нам знадобляться три двомірні матриці:

– матриці X і Y зберігатимуть координати сітки точок, в яких обчислюватиметься наведена вище функція,
– матриця Z зберігатиме значення цієї функції у відповідній точці.

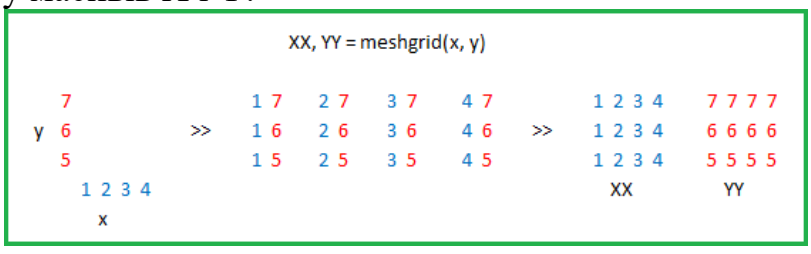
Якщо ми хочемо намалювати тривимірний графік на еквідистантній сітці (на сітці, яка має відстань між точками однакова), то для створення матриць, які зберігатимуть координати, будемо використовувати функцію `meshgrid()` з бібліотеки `numpy`.

Ця функція створює двовимірні матриці сіток за одновимірними масивами. Робота цієї функції дуже наочно показана нижче:

```
>>> X, Y = numpy.meshgrid([1,2,3], [4,5,6,7])
>>> X
array([
[1, 2, 3],
[1, 2, 3],
[1, 2, 3],
[1, 2, 3]
])

>>> Y
array([
[4, 4, 4],
[5, 5, 5],
[6, 6, 6],
[7, 7, 7]
])
```

Тепер індексу вузла сітки можна дізнатися реальні координати: `X[0][0] = 1, Y[0][0] = 4` тощо.

Фактично функція `[X, Y] = meshgrid(x, y)` задає сітку на площині **xy** у вигляді двовимірних масивів X, Y, які визначаються одновимірними масивами x та y. Рядки масиву X є копіями вектора x, а стовпці – копіями вектора y. Формування таких масивів полегшує обчислення функцій двох змінних, дозволяючи застосовувати операції над масивами. Нижче наведено малюнок, що пояснює побудову масивів X і Y:



Щоб відокремити підготовку даних від самого малювання, створення сітки та розрахунок функції виділимо в окрему функцію:

```
def makeData (): # Будуємо сітку в інтервалі від -10 до
                 # 10 з кроком 0.1 по обох координатах
    x = numpy.arange (-10, 10, 0.1)
    y = numpy.arange (-10, 10, 0.1)
# Створюємо двомірну матрицю - сітку
    xgrid, ygrid = numpy.meshgrid(x, y)
# у вузлах розраховуємо значення функції
    zgrid = numpy.sin (xgrid) * numpy.sin (ygrid) /
                          (xgrid * ygrid)
    return xgrid, ygrid, zgrid
```

Ця функція повертає три двовимірні матриці: x, y, z. Координати x та y лежать в інтервалі від -10 до 10 з кроком 0.1.

Тепер повертаємось безпосередньо до малювання. Щоб відобразити наші дані, достатньо викликати метод **`plot_surface()`** екземпляра класу Axes3D, який передаємо отримані за допомогою функції **`makeData()`** двомірні матриці.

Тепер наш приклад виглядає так:

```
import pylab
from mpl_toolkits.mplot3d import Axes3D
import numpy
def makeData():
   x = numpy.arange (-10, 10, 0.1)
   y = numpy.arange (-10, 10, 0.1)
   xgrid, ygrid = numpy.meshgrid(x, y)
   zgrid = numpy.sin (xgrid) * numpy.sin (ygrid)/  \
            (xgrid * ygrid)
   return xgrid, ygrid, zgrid

x, y, z = makeData()
fig = pylab.figure()
axes = Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(axes)

# можна "підписати" осі координат:
axes.set_xlabel("-- x -->")
axes.set_ylabel("-- y -->")
axes.set_zlabel("-- z -->")
axes.plot_surface(x, y, z)
pylab.show()
```

Якщо ми запустимо цей скрипт, то з'явиться вікно:

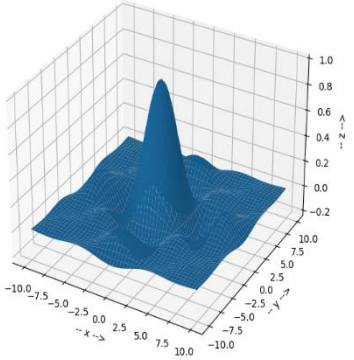


Рисунок можна обертати за допомогою мишки. `Matplotlib` не використовує графічний прискорювач, тому обертання відбувається досить повільно, хоча швидкість залежить кількості точок на поверхні.

Розглянемо інші параметри методу `Axes3D.plot_surface(X, Y, Z, args, kwargs)`, їх не так багато:

- *X*, Y та Z – ці параметри задають сітку та значення функції у вузлах.
- `rstride` та `cstride` задають крок виведення графіка. Чим менший крок, тим точніше відображається функція, але довше відбувається малювання.
- `color` – задає колір графіка
- `cmap` – задає градієнт кольорів, щоб колір комірки графіка залежав від значення функції у цій галузі.

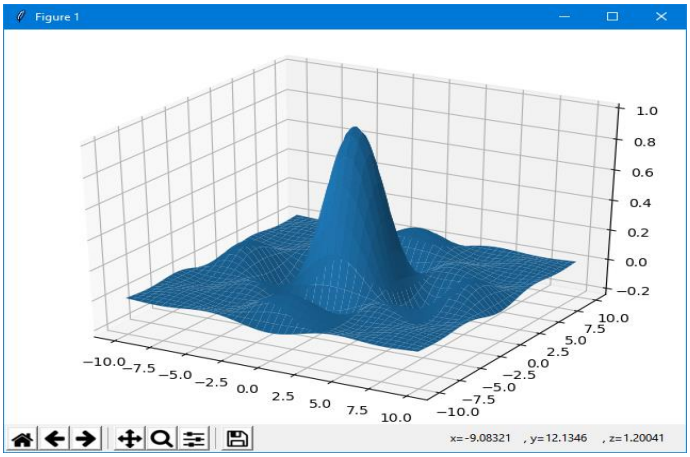
Розглянемо ці параметри.

Крок сітки.

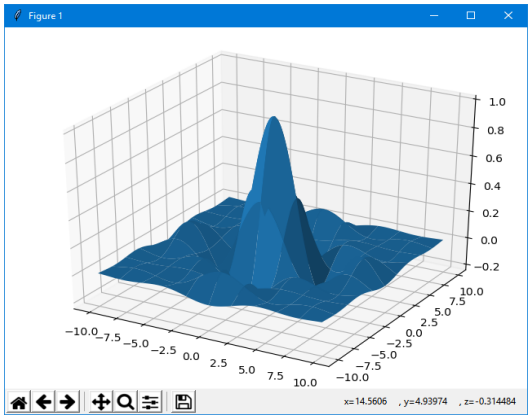
Спочатку подивимося як впливають зовнішній вигляд параметри rstride і cstride. Змінимо у попередньому прикладі рядок з використанням методу plot_surface() наступним чином:

```
axes.plot_surface(x, y, z, rstride=5, cstride=5)
```

В результаті ми отримаємо дрібнішу сітку на графіку:



Якщо так само встановити значення цих параметрів в 20, то сітка буде навпаки більша:



Зміна кольору

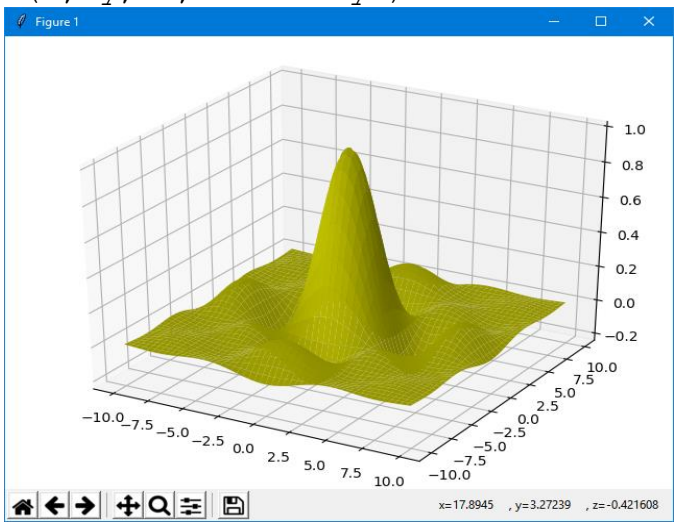
Тепер змінимо колір поверхні за допомогою параметра color. Цей параметр є рядком, який описує колір. Рядок кольору може задаватися різними способами:

Колір можна визначити англійським словом для відповідного кольору або однією літерою. Таких квітів небагато:

- 'b' або 'blue'
- 'g' або 'green'
- 'r' або 'red'
- 'c' або 'cyan'
- 'm' або 'magenta'
- 'y' або 'yellow'
- 'k' або 'black'
- 'w' або 'white'

Для прикладу зробимо поверхню жовтою:

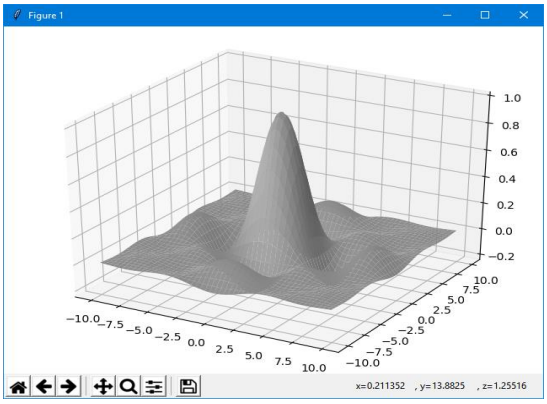
```
axes.plot_surface(x, y, z, color='yellow') або
axes.plot_surface(x, y, z, color='y')
```



Якщо потрібен сірий колір, його яскравість можна задати за допомогою рядка, що містить число в інтервалі від 0.0 до 1.0 (0 – білий, 1 – чорний). Наприклад, можна написати наступний рядок:

```
axes.plot_surface(x, y, z, color='0.7')
```

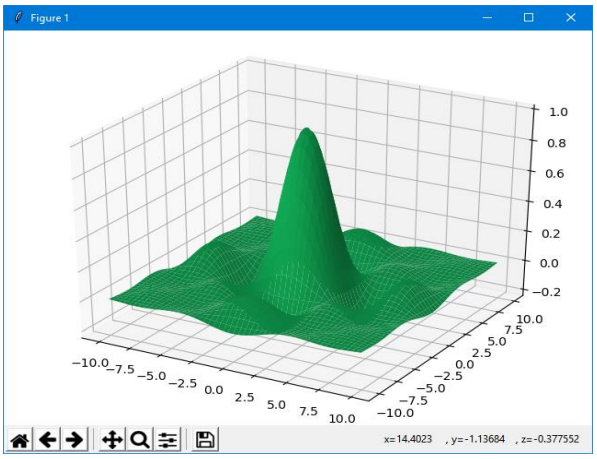
У цьому випадку побачимо такий ось сірий графік:



Крім того, можна задавати колір, оскільки це прийнято в HTML після символу решітки ('#'). Наприклад, можемо задати колір так:

```
axes.plot_surface(x, y, z, color='#11aa55')
```

Тоді графік позеленіє:



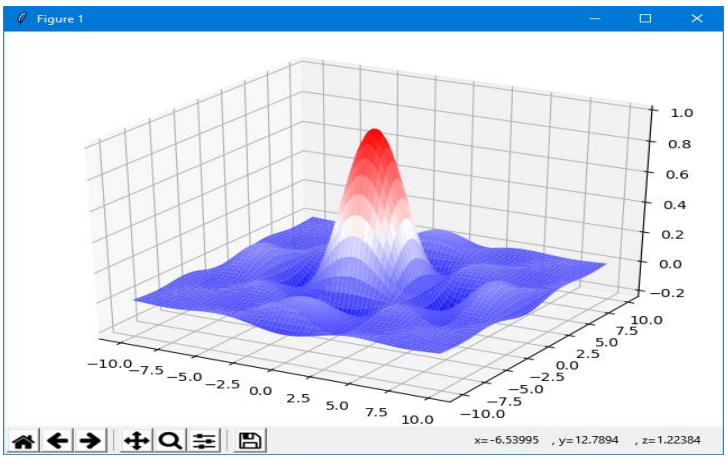
Використання колірних карток (colormap)

Кольорові карти використовуються, якщо потрібно вказати в які кольори мають фарбуватись ділянки тривимірної поверхні залежно від значення Z у цій галузі (завдання кольорового градієнта).

Щоб при виведенні графіка використовувався градієнт, як значення параметра cmap (від слова colormap, колірна карта) потрібно передати екземпляр класу `matplotlib.colors.Colormap` або похідного від нього.

Наступний приклад використовує клас `LinearSegmentedColormap`, похідний від `Colormap`, щоб створити градієнт переходу від синього кольору до червоного через білий.

```
import pylab
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LinearSegmentedColormap
import numpy
def makeData():
   x = numpy.arange (-10, 10, 0.1)
   y = numpy.arange (-10, 10, 0.1)
   xgrid, ygrid = numpy.meshgrid(x, y)
   zgrid = numpy.sin (xgrid) * numpy.sin (ygrid) /  \
                                        (xgrid * ygrid)
   return xgrid, ygrid, zgrid
x, y, z = makeData()
fig = pylab.figure()
axes = Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(axes)
axes.plot_surface(x, y, z, rstride=3,  \
       cstride=3, cmap =               \
     LinearSegmentedColormap.from_list ("red_blue",  \
                                [ 'b', 'w', 'r'], 256))
pylab.show()
```

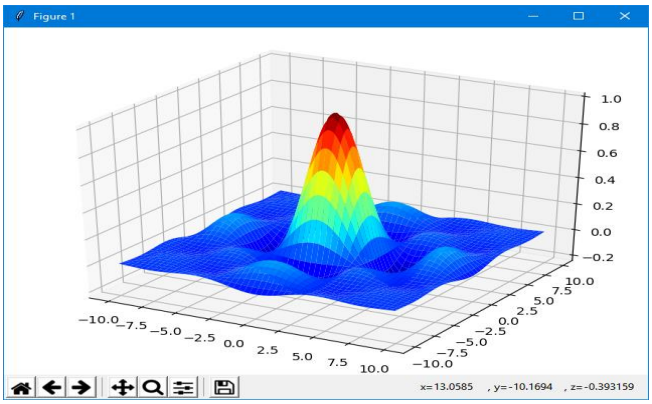


Тут використовується статичний метод **from_list(),** який приймає три параметри:

– Ім'я створюваної картки
– Список кольорів, починаючи з кольору для мінімального значення на графіці (блакитний – 'b'), через проміжні кольори (у нас це білий – 'w') до кольору для максимального значення функції (червоний – 'r').
– Кількість переходів кольорів. Чим це число більше, тим більш плавний градієнт, але тим більше пам'яті він займає.

Карта `cm.jet` – це, напевно, найпоширеніша карта в прикладах.

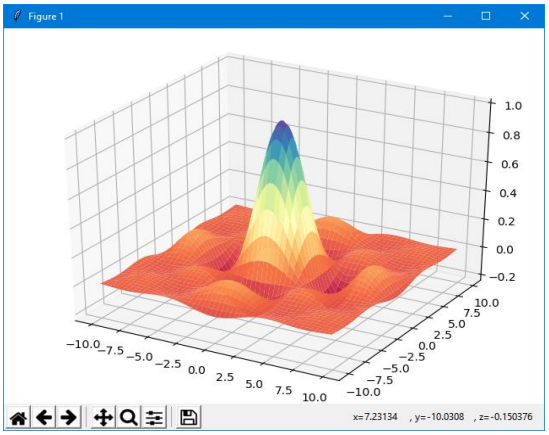
```
import pylab
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LinearSegmentedColormap
from matplotlib import cm
import numpy
def makeData():
    x = numpy.arange (-10, 10, 0.1)
    y = numpy.arange (-10, 10, 0.1)
    xgrid, ygrid = numpy.meshgrid(x, y)
    zgrid = numpy.sin (xgrid) * numpy.sin (ygrid) /  \
                                          (xgrid * ygrid)
    return xgrid, ygrid, zgrid
x, y, z = makeData()
fig = pylab.figure()
axes = Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(axes)
axes.plot_surface(x, y, z, rstride=4, cstride=4, cmap=cm.jet)
pylab.show()
```



Зауважимо, що як аргумент cmap ми передаємо не рядок, а ім'я змінної з модуля cm, причому це вже створений екземпляр класу, а не ім'я класу.

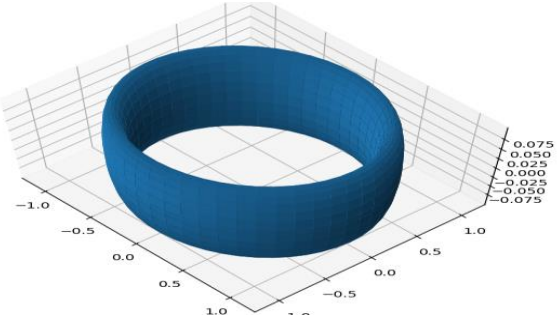
Так, наприклад, `cm.jet` – це екземпляр класу `matplotlib.colors.LinearSegmentedColormap`.

Для прикладу колірна карта cm.Spectral виглядає так:



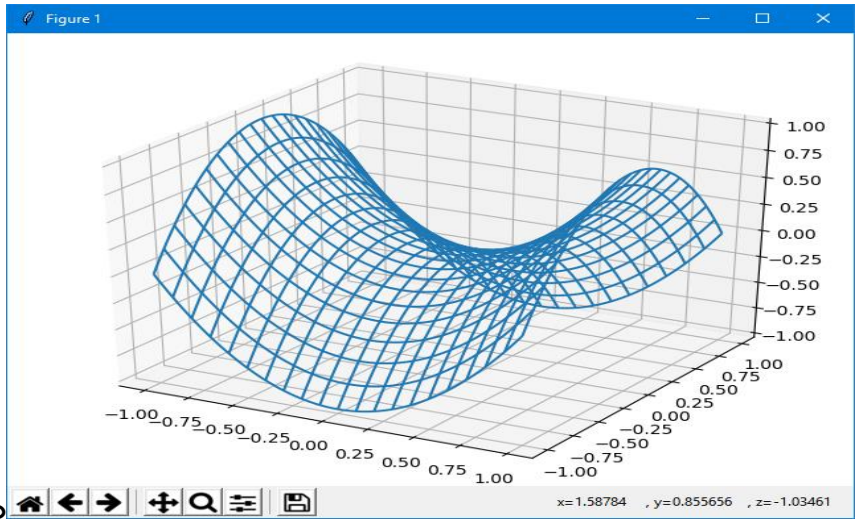
Параметричні поверхні з параметрами ϑ φ

```
import pylab
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LinearSegmentedColormap
from matplotlib import cm
import numpy as np

t=np.linspace(0,2*np.pi,50)
th,ph=np.meshgrid(t,t)
r=0.1
R=1
x,y,z=(R+r*np.cos(ph))*np.cos(th),   \
                (R+r*np.cos(ph))*np.sin(th),r*np.sin(ph)
fig=pylab.figure()
ax = Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(ax)
ax.elev=60
ax.set_aspect('equalyz')
ax.plot_surface(x,y,z,rstride=2,cstride=1)
pylab.show()
```



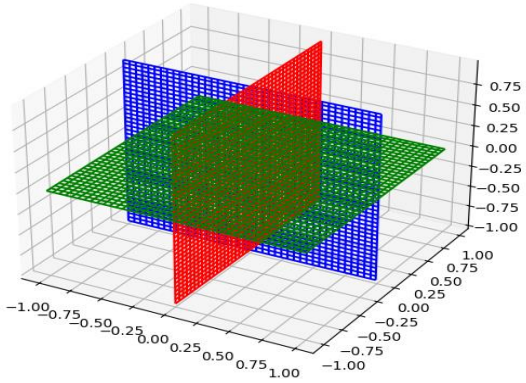
Побудова графіка «сітки» функції двох змінних

```
import pylab
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
fig=pylab.figure()
ax = Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(ax)
i = np.arange(-1, 1, 0.01)
X, Y = np.meshgrid(i, i)
Z = X**2-Y**2
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)
plt.show()
```



Для прикладу побудуємо три площини, паралельні координатним площинам.

```
import mpl_toolkits.mplot3d
from mpl_toolkits.mplot3d import Axes3D

import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
axes =  Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(axes)
t = np.arange(-1, 1, 0.01)
x, y = np.meshgrid(t, t)

axes.plot_wireframe(0*x, y, x, rstride=5, cstride=5,
                                                color='r')
axes.plot_wireframe(x, 0*y, y, rstride=5, cstride=5,
                                               color='b' )
axes.plot_wireframe(x, y, 0*x, rstride=5, cstride=5,
                                                color='g')
plt.show()
```



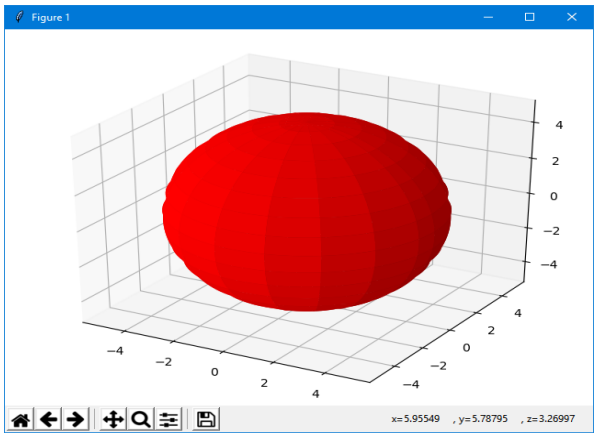
Побудова графіка функції двох змінних x=x(u,v), y=y(u,v), z=z(u,v)

Поверхня визначається параметрично: x=x(u,v), y=y(u,v), z=z(u,v). Класичне рівняння сфери, задане параметрично:

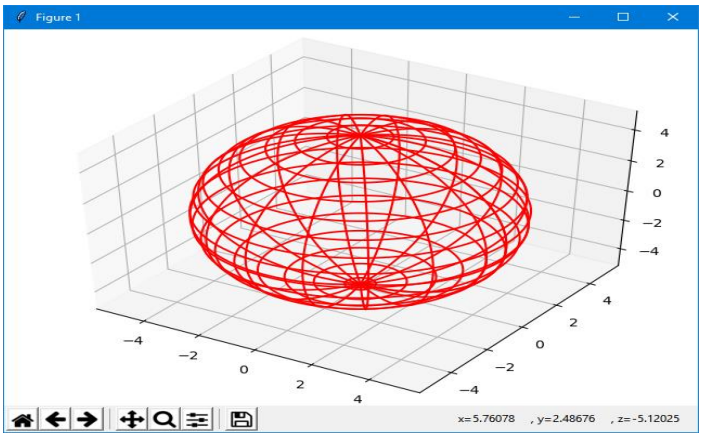
$x(u,v) = \cos(u) \cdot \cos(v)$
$y(u,v) = \sin(u) \cdot \cos(v)$
$z(u,v) = \sin(v)$
$u \in [-\pi, +\pi]$
$v \in [-2\pi, +2\pi]$

```
import pylab
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
fig=pylab.figure()
ax = Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(ax)
u = np.linspace (0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)
x = 5*np.outer(np.cos(u), np.sin(v))
y =5*np.outer(np.sin(u), np.sin(v))
z =5*np.outer(np.ones(np.size(u)), np.cos(v))
ax.plot_surface(x, y, z, rstride=6, cstride=6, color='r')
plt.show()
```

Результат роботи коду:



Замінивши `plot_surface` на `plot_wireframe` отримаємо:



У програмі використовувалася функція модуля `numpy.outer`, яка обчислює зовнішній (прямий або тензорний) добуток двох векторів:

```
numpy.outer(a, b, out=None)
```

Для двох векторів a довгою N та b довгою M зовнішній твір визначається наступним правилом:

```
[[a0*b0, a0*b1, ..., a0*bN],
[a1*b0, a1*b1, ..., a1*bN],
...
[aM*b1, aM*b1, ..., aM*bN]]
```

Це правило може бути узагальнено на масиви з більшою розмірністю. Але ця функція стискає всі багатовимірні масиви до однієї осі.

Параметри: a, b – числа, масиви `NumPy` чи подібні до масивів об'єкти. Одновимірні масиви, необов'язково однакової довжини. Двовимірні та багатовимірні масиви стискуються до однієї осі. `out` – масив `NumPy`, необов'язковий параметр.

Масив, в який можна помістити результат функції. Даний масив повинен відповідати формі та типу даних підсумкового масиву функції, а також обов'язково бути C-суміжним, тобто. зберігати дані у рядковому З стилі. Вказівка даного параметра дозволяє уникнути зайвої операції присвоювання тим самим трохи прискорюючи роботу вашого коду. Корисний параметр якщо ви часто звертаєтеся до функції в циклі.

Повертає результат – двовимірний масив NumPy, що є зовнішнім твором двох векторів.

Функція `ones()` повертає новий масив зазначеної форми та типу, заповнений одиницями. А `np.ones(np.size(u))` – повертає новий масив, розміру і так само заповнений одиницями.

У наведеному вище прикладі будувалася так звана «щільна» сітка 3-мірного координатного простору на значеннях сітки координат x, y, z відповідно. Для цього використовувалася функція прямого або тензорного добутку двох векторів значень u та v.

Можна надійти «інше» – побудувати масив щільних координатних сіток 3-мірного координатного простору для зазначених у вигляді діапазонів одновимірних масивів координатних векторів u та v. Потім «просто» застосувати формули, що задають необхідні залежності **x = x (u, v), y = y (u, v), z = z (u, v).**

Для побудови масиву щільних координатних сіток використовуємо функцію numpy.mgrid. Функція `mgrid(index object)` повертає масив щільних координатних сіток N-вимірного координатного простору для зазначених у вигляді діапазонів одновимірних масивів координатних векторів.

Параметри: `index object` – об'єкт індексації. Під об'єктом індексації розуміється список із двох або трьох елементів, наприклад [0:5] або [0:5:10j].

Якщо елементів у списку всього два, це інтерпретується як напіввідкритий інтервал `[start, ... , stop)`, у якому всі елементи відрізняються на 1, а значення stop в сам інтервал не входить.

Як третій елемент вказується уявна частина комплексного числа, яке вказує на кількість рівномірно рознесених елементів усередині закритого інтервалу `[start, ... , stop]`, при цьому значення stop потрапляє в інтервал.

Повертає: результат – масив NumPy, що є масивом щільних координатних сіток N-мірного координатного простору, кількість та розміри яких залежать від зазначених діапазонів.

Побудову виконаємо наступним скриптом:

```
import pylab
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
#fig=pylab.figure()
kg=5; kv=5
fig = plt.figure(figsize=plt.figaspect(1/kg)*kv)
ax = Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(ax)
u, v = np.mgrid[0:2 * np.pi: 100j, 0:2 * np.pi: 100j]
x = 5*np.cos(u)*np.sin(v)
y = 5*np.sin(u)*np.sin(v)
z = 5 * np.cos (v)
#axes.plot_surface(x, y, z,rstride=6, cstride=6, color='r')
ax.plot_wireframe(x, y, z, rstride=5, cstride=5, color='r')
plt.show()
```

Для отримання масиву щільних координатних сіток можна також скористатися (розглянутою вище) функцією np.meshgrid:

```
u1=np.linspace(0,2*np.pi,100)
v1 = np.linspace (0,2 * np.pi, 100)
u, v = np.meshgrid(u1,v1)
```

За потреби можна весь рисунок «розтягнути» в kg разів по горизонталі та в kv разів по вертикалі. Для цього укажемо параметр методу `plt.figure`():

```
kg=2; kv=1
fig = plt.figure(figsize=plt.figaspect(1/kg)*kv)
```

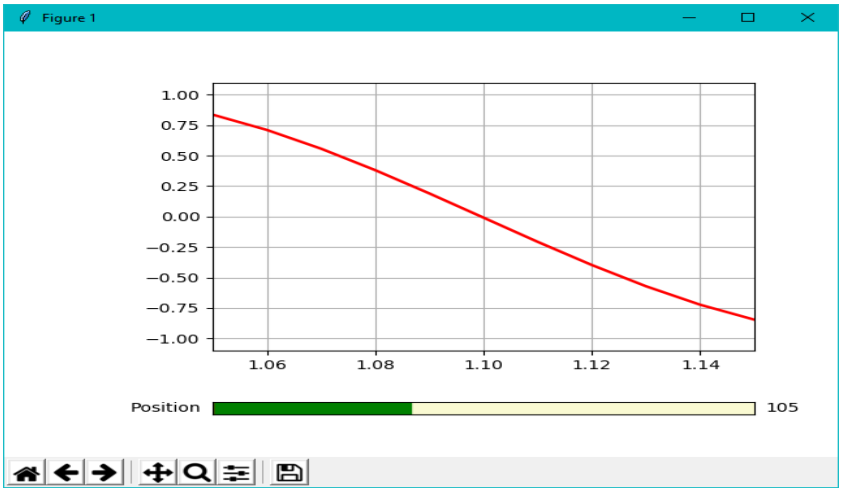
Скролінг по осі X

`Matplotlib` включає дуже невелику кількість елементів управління, які в термінах `Matplotlib` називаються віджетами, які розташовуються пакеті `matplotlib.widgets`. Цей пакет містить також віджети для взаємодії з графіками (віджети для виділення областей).

Використовуючи віджет Slider можна «розтягнути» вісь X:

```
from matplotlib.widgets import Slider
import math
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
plt.subplots_adjust(left=0.25, bottom=0.25)
t = [i / 100. for i in range(0, int(math.pi) * 100, 1)]
s = [math.sin(i * 20) for i in t]
l, = plt.plot (t, s, lw = 2, color = 'red')
plt.axis([0, 1, -10, 10])
plt.grid(True)
axcolor = 'lightgoldenrodyellow' #'gray'
ax_x_pos = plt.axes([0.25, 0.1, 0.65, 0.03], facecolor=axcolor)
wsize = 10
x_pos =  Slider(ax_x_pos, 'Position', 0,
                    len(t) - wsize - 1, valfmt='%d', \
      valinit=0, color="g")
ax.set_xlim(t[0], t[wsize])
ax.set_ylim(-1.1, 1.1)
def update(val):
    #print(x_pos.val)
    pos = int(x_pos.val)
    ax.set_xlim(t[pos], t[pos + wsize])
    fig.canvas.draw_idle()
x_pos.on_changed(update)
plt.show()
```

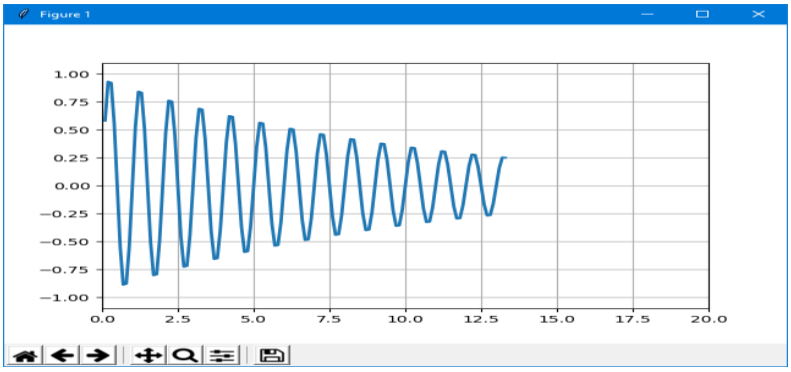
Результат роботи скрипту:

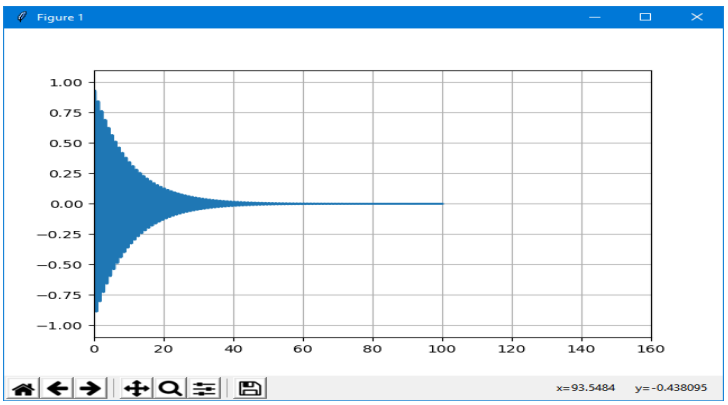


«Динамічні» графіки

Без коментарів наведемо приклад «динамічної» побудови графіка

```
'''
This example showcases a sinusoidal decay animation.
'''
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
def data_gen(t=0):
    cnt = 0
    while cnt < 1000:
        cnt += 1
        t += 0.1
        yield t, np.sin(2*np.pi*t) * np.exp(-t/10.)
def init():
    ax.set_ylim(-1.1, 1.1)
    ax.set_xlim(0, 10)
    del xdata[:]
    del ydata[:]
    line.set_data(xdata, ydata)
    return line,
fig, ax = plt.subplots()
line, = ax.plot([], [], lw=2)
ax.grid()
xdata, ydata = [], []
def run(data):
    # update the data
    t, y = data
    xdata.append(t)
    ydata.append(y)
    xmin, xmax = ax.get_xlim()
    if t >= xmax:
        ax.set_xlim(xmin, 2*xmax)
        ax.figure.canvas.draw()
    line.set_data(xdata, ydata)
    return line,
ani = animation.FuncAnimation(fig,
            run, data_gen, blit=False, interval=10,
                  repeat=False, init_func=init)
plt.show()
```





Питання для самоконтролю до теми 3

1. Яке основне призначення бібліотеки `Matplotlib`?
2. Яке призначення клавіш в інтерактивному вікні виведення графіків?
3. Яка функція бібліотеки `matplotlib` виконує побудову двовимірного графіка?
4. Як побудувати 2D графік, заданий параметрично?
5. Як побудувати тривимірний графік?

Завдання до теми 3

Сформулюйте та візуалізуйте за допомогою бібліотеки `Matplotlib` набір даних, що містить інформацію про місячні продажі трьох різних товарів (A, B, C) протягом шести місяців. Рішення повинно включати: створення лінійного графіка (`plot()`) для відображення динаміки продажів кожного товару окремою лінією; додавання заголовка графіка; підписів осей X та Y; використання легенди (`legend()`) для ідентифікації кожного товару; та зміну кольору або стилю лінії для товару C для його виділення.

4. МОДУЛЬ MATH

Вбудований модуль `math` у Python надає набір функцій для виконання математичних, тригонометричних та логарифмічних операцій.

Деякі з основних функцій модуля:

- `pow(num, power)`: зведення числа num у ступінь power
- `sqrt(num)`: квадратний корінь числа num
- `ceil(num)`: округлення числа до найближчого найбільшого цілого
- `floor(num)`: округлення числа до найближчого найменшого цілого
- `factorial(num)`: факторіал числа
- `degrees(rad)`: переведення з радіан у градуси
- `radians(grad)`: переведення з градусів у радіани
- `cos(rad)`: косинус кута в радіанах.
- `sin(rad)`: синус кута в радіанах
- `tan(rad)`: тангенс кута в радіанах
- `acos(rad)`: арккосинус кута в радіанах
- `asin(rad)`: арксинус кута в радіанах
- `atan(rad)`: арктангенс кута в радіанах
- `log(n, base)`: логарифм числа n на основі base
- `log10(n)`: десятковий логарифм числа n

Приклад застосування деяких функцій:

```
import math
# Піднесення числа 2 у ступінь 3
n1 = math.pow(2, 3)
print(n1) # 8
# ту ж саму операцію можна виконати так
n2 = 2**3
print(n2)
# Піднесення до квадрату
print(math.sqrt(9)) # 3
# найближче найбільше ціле число
print(math.ceil(4.56)) # 5
найближче найменше ціле число
print(math.floor(4.56)) # 4
# Переведення з радіан в градуси
print(math.degrees(3.14159)) # 180
# Переведення з градусів в радіани
print(math.radians(180)) # 3.1415.....
# косинус
print(math.cos(math.radians(60))) # 0.5
# Синус
print(math.sin(math.radians(90))) # 1.0
# тангенс
print(math.tan(math.radians(0))) # 0.0
# логарифм
print(math.log(8,2)) # 3.0
print(math.log10(100)) # 2.0
```

```
x=math.asin( math.sin( math.radians(30) ) )
print(x)
0.5235987755982988
print(math.degrees(x) )
29.999999999999996
```

Робота функцій pow (не з модуля `math`) з комплексними значеннями:

```
z=pow(-0.1,0.23)
print(z)
(0.4416981441638905+0.38940929610720204j)
```

Також модуль `math` надає ряд вбудованих констант, такі як, `math.pi` та `math.e`:

```
import math
radius = 30
#площа кола з радіусом 30
area = math.pi * math.pow (radius, 2)
print(area)
# натуральний логарифм числа 10
number = math.log(10, math.e)
print(number)
```

Питання для самоконтролю до теми 4

1. Яке призначення модуля `math`?
2. Якою функцією модуля `math` можна звести число в заданий ступінь?
3. Якою функцією модуля `math` можна виконати округлення числа до найближчого цілого?
4. Якою функцією модуля `math` можна знайти факторіал заданого числа?
5. Як знайти логарифм числа на основі п'ять з використанням функцій модуля `math`?

Завдання до теми 4

Напишіть скрипт виведення таблиці значень факторіалу чисел від 2 до заданого. Обчислення значення факторіалу числа виконайте двома способами - з використанням модуля `math` та функцією користувача.

5. МОДУЛЬ `fractions`: РОБОТА ІЗ РАЦІОНАЛЬНИМИ ЧИСЛАМИ

Модуль `fractions` дозволяє виконувати арифметичні дії над раціональними числами (зі звичайними дробами), що у деяких ситуаціях буває надзвичайно зручно.

Створення звичайних дробів

Способів створення екземплярів раціональних чисел можна розділити на наступні групи.

Найпростіший спосіб створити звичайний дріб – вказати чисельник (numerator) та знаменник (denominator):

```
>>> from fractions import Fraction
>>>
>>> Fraction() # за замовчуванням numerator=0, denominator=1
Fraction(0, 1)
>>>
>>> Fraction(numerator=1, denominator=2) # рівносильно Fraction(1,
2)
Fraction(1, 2)
>>>
>>> Fraction(1, 2)
Fraction(1, 2)
```

Якщо вказані чисельник і знаменник мають спільні дільники, перед створенням раціонального числа вони скорочені:

```
>>> Fraction(8, 16), Fraction(15, 30)
(Fraction(1, 2), Fraction(1, 2))
```

Як чисельник і (або) знаменник можуть бути зазначені інші дроби:

```
>>> Fraction(3, Fraction(1, 2))
Fraction(6, 1)
>>>
>>> Fraction(Fraction(1, 2), 3)
Fraction(1, 6)
>>>
>>> Fraction(Fraction(1, 2), Fraction(3, 7))
Fraction(7, 6)
```

Ціле і речове число, так само можна перетворити на звичайний дріб:

```
>>> Fraction(10)
Fraction(10, 1)
>>>
>>> Fraction(11.11)
Fraction(781796747813847, 70368744177664)
>>>
>>> Fraction(1.25)
Fraction(5, 4)
>>>
```

```
>>> Fraction(2e-10)
Fraction(7737125245533627, 38685626227668133590597632)
```

А ось комплексні числа призведуть до помилки:

```
>>> Fraction(1j, 1 + 2j)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: both arguments повинні бути Rational instances
```

Але уявною і дійсною частиною комплексного числа можуть бути вказані раціональні числа:

```
>>> complex(Fraction(3, 5), Fraction(4, 7))
(0.6+0.5714285714285714j)
```

Так само будь-який десятковий дріб модуля Decimal може бути перетворений на звичайний дріб:

```
>>> з import decimal Decimal
>>> Fraction(Decimal('7.7'))
Fraction(77, 10)
```

Будь-який рядок, який може бути перетворений на будь-яке допустиме число, також може бути використаний для отримання звичайних дробів:

```
>>> Fraction('111')
Fraction(111, 1)
>>>
>>> Fraction('1.11')
Fraction(111, 100)
>>>
>>> Fraction('1.11e+10')
Fraction(11100000000, 1)
>>>
>>> Fraction('-17/63')
Fraction(-17, 63)
>>>
>>> Fraction('-0.115')
Fraction(-23, 200)
>>>
>>> Fraction('-.115')
Fraction(-23, 200)
>>>
>>> Fraction('1.11 \t\n')
Fraction(111, 100)
>>> Fraction('\t\n 1.11 \t\n')
Fraction(111, 100)
>>> Fraction('\t\n 1.11')
Fraction(111, 100)
>>>
>>> Fraction('\t\n -13/37')
Fraction(-13, 37)
```

Математичні операції над раціональними числами

Усі арифметичні оператори підтримують обчислення з раціональними числами:

```
>>> x = Fraction(2, 5)
>>> y = Fraction(3, 7)
>>>
>>> x + y, y - x
(Fraction(29, 35), Fraction(1, 35))
>>>
>>> x*y, x/y
(Fraction(6, 35), Fraction(14, 15))
>>>
>>> x**0.5, 2**0.5/5**0.5
(0.6324555320336759, 0.6324555320336759)
>>>
>>> x**y, (x**3)**(1/7)
(0.6752339686501552, 0.6752339686501552)
>>>
>>> y//x
1
>>> x%y
Fraction(2, 5)
```

Однак арифметичні оператори не здатні до дій над типами `Fraction` та `decimal.Decimal` в одному виразі:

```
>>> x + 1.1
1.5
>>>
>>> x + Decimal('1.1') # призведе до помилки
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Fraction' and
'decimal.Decimal'
```

Функції модуля math здатні приймати раціональні числа як аргументи, так як останні, по суті, можуть бути просто перетворені до речовинних чисел:

```
>>> import math
>>>
>>> x = Fraction(4, 11)
>>>
>>> math.exp(x), math.exp(4/11)
(1.438551009577678, 1.438551009577678)
```

`Fraction` – атрибути та методи

`x.numerator`

повертає чисельник дробу:

```
>>> from fractions import Fraction
>>>
```

```
>>> x = Fraction(3, 8)
>>> x
Fraction(3, 8)
>>>
>>> x.numerator
3
```

x.denominator

повертає знаменник дробу:

```
>>> x.denominator
8
```

Fraction.from_float(flt)

приймає `flt` – число типу `float` і повертає звичайний дріб ставлення чисельника до знаменника якого максимально наближається до значення `flt`:

```
>>> from fractions import Fraction
>>>
>>> Fraction.from_float(0.5)
Fraction(1, 2)
>>>
>>> Fraction.from_float(0.6)
Fraction(5404319552844595, 9007199254740992)
```

Як можна помітити, через зберігання чисел з рухомою комою в двійковому поданні `Fraction.from_float(0.6)` зовсім не одно `Fraction(6, 10)`:

```
>>> Fraction.from_float(0.6) == Fraction(6, 10)
False
```

Fraction.from_decimal(dec)

створює звичайний дріб, який є точним уявленням десяткового дробу зазначеного в `dec`, де `dec` – це екземпляр класу decimal.

```
>>> from import decimal Decimal
>>>
>>> Fraction.from_decimal(Decimal('0.7'))
Fraction(7, 10)
>>>
>>> Fraction.from_decimal(Decimal('0.77'))
Fraction(77, 100)
>>>
>>> Fraction.from_decimal(Decimal('0.125'))
Fraction(1, 8)
```

Fraction.limit_denominator(max_denominator=1000000)

повертає найближче раціональне уявлення вказаного числа, знаменник якого не перевищує значення max_denominator.

```
>>> import math
>>> math.pi
3.141592653589793
>>>
>>> Fraction(math.pi).limit_denominator(10)
```

```
Fraction(22, 7)
>>>
>>> Fraction(math.pi).limit_denominator(1000)
Fraction(355, 113)
>>>
>>> Fraction(math.pi).limit_denominator(100000)
Fraction(312689, 99532)
```

Даний метод дозволяє отримувати як раціональні наближення, так і відновлювати раціональні значення за їх неточним поданням у вигляді чисел з рухомою комою:

```
>>> math.cos(math.pi/3) # точне значення 0.5
0.5000000000000001
>>>
>>> Fraction(math.cos(math.pi/3)).limit_denominator()
Fraction(1, 2)
>>>
>>>
>>> 2**(1/3)
1.2599210498948732
>>>
>>> Fraction(2**(1/3)).limit_denominator()
Fraction(1054215, 836731)
>>>
>>> 1054215/836731
1.2599210498953666
```

x.__floor__()

повертає значення типу int, яке є найбільшим серед усіх int <= x:

```
>>> Fraction(234, 47).__floor__()
4
```

Та й з огляду на те, що всі функції модуля math можуть обробляти прості дроби, то скористатися даним методом можна і так:

```
>>> math.floor(Fraction(234, 47))
4
```

x.__ceil__()

повертає значення типу int, яке є найменшим серед усіх int >= x:

```
>>> Fraction(234, 47).__ceil__()
5
>>> import math
>>> math.ceil(Fraction(234, 47))
5
```

x.__round__(ndigits=None)

Округлює до найближчого парного числа.

Якщо `ndigits=None`, округлення виконується до найближчого парного цілого числа:

```
>>> Fraction('1/2').__round__()
```

```
0
>>>
>>> Fraction('3/2').__round__()
2
```

Якщо ndigits не дорівнює None, то округлення виконується до найближчого числа кратного дробу `Fraction(1, 10**ndigits)`, при цьому якщо остання цифра 5 то округлення буде виконано до найближчої парної цифри:

```
>>> Fraction('7/3').__round__(2)
Fraction(233, 100)
>>>
>>> Fraction('7/3').__round__(3)
Fraction(2333, 1000)
>>>
>>> Fraction('555/1000').__round__(1)
Fraction(3, 5)
>>>
>>> Fraction('555/1000').__round__(2)
Fraction(14, 25)
```

Якщо ndigits не дорівнює `None`, але менше 0, то округлення виконується до розряду на який вказує `abs(ndigits)`, при цьому якщо остання цифра 5 то округлення буде виконано до найближчої парної цифри:

```
>>> Fraction('-5555555/10').__round__(-1)
Fraction(-555560, 1)
>>> Fraction('-5555555/10').__round__(-3)
Fraction(-556000, 1)
>>>
>>> Fraction('77777/10').__round__(-3)
Fraction(8000, 1)
```

`fractions.gcd(a, b)`

повертає найбільший загальний дільник чисел a та b.

Повертає НСД(a, b). Знак результату залежить від знака b, якщо b не дорівнює 0, інакше знак результату дорівнює знаку a. Повертає 0, тільки якщо a та b обидва рівні 0:

```
>>> fractions.gcd(135, 15)
15
>>> fractions.gcd(135, -15)
-15
>>> fractions.gcd(-135, 15)
15
>>> fractions.gcd(-135, 0)
-135
>>> fractions.gcd(0, 0)
0
```

З версії 3.5. вважається застарілим, і краще використовувати команду math.gcd().

Приклад

Обчислити $2:\frac{3}{5}+\frac{3}{5}:2+1\frac{1}{2}:6+6:1\frac{1}{2}$

```
from fractions import Fraction as dr
rez=dr(2,1)/dr(3,5)+dr(3,10)+dr(3,2)/6+6/dr(3,2)
print(rez, '=',end="")
a = int (rez.numerator / rez.denominator)
b = rez - a
print(a,"+", b, sep="")

>>> 473/60 = 7+53/60
```

Питання для самоконтролю до теми 5

1. Як створити звичайний дріб для роботи з функціями модуля fractions?
2. При створенні дробу чисельники чи знаменник створюваного дробу бути дробом?
3. Чи функції модуля fractions підтримують роботу з комплексними числами?
4. Які математичні операції підтримуються функціями та перевантаженими операторами модуля fractions?
5. Які основні методи (функції) є в модулі fractions?

Завдання до теми 5

Необхідно розробити функцію побудови об'єкта класу раціонального числа за значенням цілої частини дробу та значення чисельника та знаменника дробу. Розробляйте функцію перетворення раціонального дробу на цілу частину та «правильний дріб».

6. МОДУЛЬ `cmath`

Модуль `cmath` надає функції для роботи з комплексними числами.

`cmath.phase(x)` – повертає фазу комплексного числа (її ще називають аргументом). Еквівалентно `math.atan2(x.imag, x.real)`. Результат лежить у проміжку [-π, π]. Отримати модуль комплексного числа можна за допомогою вбудованої функції `abs()`.

`cmath.polar(x)` – перетворення до полярних координат. Повертає пару `(r, phi)`.

`cmath.rect(r, phi)` – перетворення з полярних координат.

`cmath.exp(x)` – `exp`.

`cmath.log(x[, base])` – логарифм x на основі `base`. Якщо `base` не вказано, повертається натуральний логарифм.

`cmath.log10(x)` – десятковий логарифм.

`cmath.sqrt(x)` – квадратний корінь із x.

`cmath.acos(x)` – арккосинус x.

`cmath.asin(x)` – арксинус x.

`cmath.atan(x)` – арктангенс x.

`cmath.cos(x)` – косинус x.

`cmath.sin(x)` – синус x.

`cmath.tan(x)` – тангенс x.

`cmath.acosh(x)` – гіперболічний арккосинус x.

`cmath.asinh(x)` – гіперболічний арксинус x.

`cmath.atanh(x)` – гіперболічний арктангенс x.

`cmath.cosh(x)` – гіперболічний косинус x.

`cmath.sinh(x)` – гіперболічний синус `x`.

`cmath.tanh(x)` – гіперболічний тангенс `x`.

`cmath.isfinite(x)` – `True`, якщо дійсна та уявна частини кінцеві.

`cmath.isinf(x)` – `True`, якщо дійсна, або уявна частина нескінченна.

`cmath.isnan(x)` – `True`, якщо або дійсна, або уявна частина `NaN`.

`cmath.pi` – π.

`cmath.e` – e.

Приклади.

```
import cmath
z=1+1j
print('z=',z)
z1=cmath.polar(z)
print('z1=',z1)
import math
print(z1[0], math.degrees(z1[1]) )
x=-1+0j
```

```
z2=cmath.sqrt(x)
print(f"x={x} sqrt(x)={z2}")
x=0+1j
z2=x **2
print(f"x={x} x**2={z2}")

#  тотожність Ейлера e^(ix)=cos(x)+isin(x}
a=0+math.pi*1j
z=cmath.exp(a)
print(f"exp(math.pi*1j = {z}")

#ln(cos(x) + isin(x)=ix
z=cmath.log(cmath.cos(2) + cmath.sin(2)*1j)
print(z)
```

Результат роботи програми

```
z= (1+1j)
z1= (1.4142135623730951, 0.7853981633974483)
1.4142135623730951 45.0
x=(-1+0j) sqrt(x)=1j
x=1j x**2=(-1+0j)
exp(math.pi*1j = (-1+1.2246467991473532e-16j)
(1.3877787807814457e-17+2j)
```

Питання для самоконтролю до теми 6

1. Як визначити фазу комплексного числа?
2. Як знайти квадратний корінь із негативного числа?
3. Як знайти гіперболічний синус?
4. Як знайти логарифма комплексного числа з негативної основи?
5. Як знайти модуль комплексного числа?

Завдання до теми 6

Напишіть скрипт зображення векторів у декартовій системі координат відповідних заданим комплексним числам. Задані комплексні числа передаються у програму як кортеж.

7. МОДУЛЬ `struct` – ЗАПИС ДАНИХ У БІНАРНИЙ ФАЙЛ

У деяких додатках доводиться мати справу з упакованими двійковими даними, які створюються, наприклад, програмами на мові С. Для їх збереження та відновлення можна скористатися модулем `struct` із стандартної бібліотеки Python.

Приклад запису у файл та читання з файлу:

```
import struct

myfile = open("data.bin", "wb")
# упаковуємо дані
bytes = struct.pack(b'>i4sh', 7, b'spam', 8)
myfile.write(bytes)
myfile.close()

myfile = open("data.bin", "rb")
data = myfile.read()
# Вилучаємо дані
values = struct.unpack(b'>i4sh', data)
print(values)
```

Методи

`struct.unpack(format, data)`

Розпаковує дані зі структури

Приклади:

```
два цілі 4х байтові числа прямого порядку
struct.unpack('>LL', b'x00x00x00x9ax00x00x00x8d')
# (154, 141)

два цілі 4х байтові числа прямого порядку
struct.unpack('>2L', b'x00x00x00x9ax00x00x00x8d')
# (154, 141)

два цілі 4х байтові числа прямого порядку, пропустивши 1 і 2 байти
по краях
struct.unpack('>1x2L2x',
 b'x00x00x00x00x9ax00x00x00x8dx00x00')
# (154, 141)
```

`struct.pack(format, data)`

Пакує дані до структури

```
struct.pack('>L', 154)
# b'x00x00x00x9a'

struct.pack('>L', 141)
# b'x00x00x00x8d'
```

Специфікатори формату

> Прямий порядок
< –зворотний порядок
x– пропустити один байт
b –знаковий один байт
B –беззнаковий байт
h –знакове коротке ціле число, 2 байти
H –беззнакове коротке ціле число, 2 байти
i –знакове ціле число, 4 байти
I –беззнакове ціле число, 4 байти
l –знакове довге ціле число, 4 байти
L –беззнакове довге ціле число, 4 байти
Q –беззнакове дуже довге ціле число, 8 байт
f –число з рухомою комою, 4 байти
d –число з рухомою комою подвійної точності, 8 байт
p –лічильник та символи, 1 + count байт
s –символи, count символів

Приклад запису та читання даних у бінарному файлі

```
import sys
import struct
myfile = open("data.bin", "wb")
# упаковуємо дані
for i in range(100):
  x1=10.0*float(i+0)
  x2=10.0*float(i+1)
  x3=10.0*float(i+2)
  x4=10.0*float(i+3)
  x5=10.0*float(i+4)
  print(x1, x2, x3, x4, x5)
  b=struct.pack(b'>fffff', x1,x2,x3,x4,x5)
  myfile.write(b)
myfile.close()
#sys.exit(0)
myfile = open("data.bin", "rb")
while (True):
  data = myfile.read(4*5)
  if data==b'':
    break
  # Вилучаємо дані
  values = struct.unpack(b'>fffff', data)
  y1=values[0]
  y2=values[1]
  y3=values[2]
  y4=values[3]
  y5=values[4]
  print(y1, y2, y3, y4, y5)
myfile.close()
```

Приклад запису файлу на FORTRAN та читання на PYTHON даних у бінарному файлі

Формування файлу:

```
! компілятор ming gfortran
 implicit none
 integer :: j, i
 real(4), DIMENSION(5) :: buf
 character(40) :: fname='d:\float5.bin'
 open(unit=50, file=trim(fname), status='UNKNOWN',
form='UNFORMATTED',err=100)
!
! запис у файлі містить у "початку та "в кінці" запису
! "дескриптор" Чотири байти - кількість байт
! у запису крім дескрипторів
!
 do i=1,50
 buf(1)=float(i)
 buf(2)=float(i+1)
 buf(3)=float(i+2)
 buf(4)=float(i+3)
 buf(5)=float(i+4)
 write(50) (buf(j), j=1,5)
 write(*, '( 5(g12.5,1x) )' ) (buf(j), j=1,5)
 enddo
 close(50)
 stop 0
100 write(*,*) '* error open file ', trim(fname), '*'
 stop 16
 end
```

Читання файлу:

```
import sys
import struct
myfile = open(r"d:\float5.bin", "rb")
##kbyteB=myfile.read(4) # дескриптор запису
nzap=0
while(True):
  kbyteB=myfile.read(4) # дескриптор запису -
  # кількість байт у цьому записі
  if kbyteB==b'':
    break
  nzap+=1
  kbyte = struct.unpack(b'<L', kbyteB)[0]
  print('запис № ',nzap,' містить ',kbyte,' байт')
  data = myfile.read(4*5) #читання запису
  if data==b'':
    break
  # Вилучаємо дані
  values = struct.unpack(b'<fffff', data)
  y1=values[0]
  y2=values[1]
```

```
  y3=values[2]
  y4=values[3]
  y5=values[4]
  print(y1, y2, y3, y4, y5) #values)
  kbyteB=myfile.read(4) # дескриптор кінця запису -
                        # кількість байт у цьому записі
myfile.close()
```

Питання для самоконтролю до теми 7

1. Яке основне призначення програм модуля `struct`?
2. Які основні методи модуля `struct` використовуються для роботи з бінарними структурами?
3. Які специфікатори формату використовуються для перетворення даних у модулі `struct`?
4. Як прочитати дані з бінарного файлу, створеного іншими програмами?
5. Що таке дескриптор бінарного файлу?

Завдання до теми 7

Напишіть скрипт виведення на консоль байт бінарного файлу як таблиці - дампа. Таблиця повинна містити три стовпця: перший – значення усунення кількості байт від початку файлу. Другий стовпець повинен містити звільнене уявлення чергових 16-ти байт файлу у форматі запису 16-х чисел, записаних через пробіл. Третій стовпець – символи, що відповідають байтам другого стовпця. Якщо байт не є друкованим символом, замість нього необхідно виводити символ крапки (.).

8. ФАЙЛИ CSV

Програмісти часто стикаються із завданням обробки великих обсягів структурованих даних. Python має вбудовану бібліотеку CSV, за допомогою якої програміст може працювати із спеціальними CSV файлами. Це своєрідні електронні таблиці.

Кожен рядок у файлі csv представляє окремий запис або рядок, який складається з окремих стовпців, розділених комами. Саме тому формат і називається Comma Separated Values. Але хоча формат CSV – це формат текстових файлів, Python для спрощення роботи з ним надає спеціальний вбудований модуль CSV

Що таке файли CSV

CSV – це особливий вид файлу, який дозволяє структурувати великі обсяги даних.

По суті він є звичайним текстовим файлом, однак кожен новий елемент відокремлений від попереднього комою або іншим роздільником. Зазвичай кожен запис починається з нового рядка. Дані CSV можна легко експортувати до електронних таблиць або баз даних. Програміст може розширювати файл CSV, додаючи нові рядки.

Приклад CSV файлу, де як роздільник використовується кома:

```
Ім'я,Професія,Рік народження
Віктор, Токар, 1995
Сергій, Зварювальник, 1983
```

Як видно з прикладу, у першому рядку зазвичай вказується, яка інформація буде знаходитись у кожному стовпці. Крім того, після останнього елемента рядка кома не ставиться, інтерпретатор визначає кінець рядка за символом перенесення.

Замість коми можна використовувати будь-який інший роздільник, тому при читанні файлу CSV потрібно заздалегідь знати, який символ використовується.

Важливо пам'ятати, що CSV – це звичайний текстовий файл, який не підтримує символи в кодуваннях, що відрізняються від ASCII або Unicode.

Бібліотека CSV

Це основна бібліотека для роботи з файлами CSV в Python.

Бібліотека csv є вбудованою, тому її не потрібно завантажувати, достатньо використовувати звичайний імпорт:

```
import csv
```

Читання з файлів (парсинг)

Для того, щоб прочитати дані з файлу, програміст повинен створити об'єкт reader:

```
reader_object = csv.reader(file, delimiter = ",")
```

`reader` має метод __next__(), тобто є об'єктом, що ітерується, тому читання з файлу відбувається наступним чином:

```
import csv
with open("classmates.csv", encoding='utf-8') as r_file:
   # Створюємо об'єкт reader,
   #                  вказуємо символ-розділювач ","
    file_reader = csv.reader(r_file, delimiter = ",")
   # Лічильник для підрахунку кількості
   # рядків та виведення заголовків стовпців
    count = 0
   # Зчитування даних із CSV файлу
    for row in file_reader:
        if count == 0:
       # Виведення рядка, що містить
       #                          заголовки для стовпців
            print(f' Файл містить стовпці: {",
                                        ".join(row)}')
        else:
    # Виведення рядків
            print(f'{row[0]} - {row[1]} і він
                         народився в {row[2]} році.')
        count += 1
    print(f' Всього у файлі {count} рядків.')
```

Припустимо, що ми маємо CSV файл, який містить таку інформацію:

```
Ім'я,Успішність,Рік народження
Саша, відмінник, 200
Маша,хорошистка,1999
Петя, трієчник, 2000
```

Тоді, якщо відкрити цей файл у нашій програмі, будуть отримані наступні результати:

```
Файл містить стовпці: Ім'я, Успішність, Рік народження
 Сашко – відмінник і він народився 200 року.
 Маша – хорошистка і він народився 1999 року.
 Петро – трієчник і він народився 2000 року.
Усього у файлі 4 рядків.
```

Використання конструкції `with…as` дозволяє програмісту бути впевненим, що файл буде закритий, навіть якщо під час виконання коду станеться якась помилка.

Зверніть увагу, що при відкритті потрібно вказати правильне кодування, в якому зберігаються дані. В даному випадку `encoding = 'utf-8'`. Якщо не вказувати, то використовуватиметься кодування за замовчуванням. Для Windows це CP1251.

Бібліотека CSV дозволяє працювати з файлами, як із словниками, для цього потрібно створити об'єкт `DictReader`. Звертатися до елементів можна на ім'я стовпців, а не за допомогою індексів. Щоб вихідна програма робила аналогічний висновок, її слід змінити так:

```
import csv
with open("classmates.csv", encoding='utf-8') as r_file:
   # Створюємо об'єкт DictReader,
   # вказуємо символ-розділювач ","
    file_reader = csv.DictReader(r_file, delimiter = ",")
   # Лічильник для підрахунку кількості
   # рядків та виведення заголовків стовпців

    count = 0
   # Зчитування даних із CSV файлу
    for row in file_reader:
      if count == 0:
       # Виведення рядка, що містить заголовки
       # для стовпців
       print(f'Файл містить стовпці: {", ".join(row)}')
       # Виведення рядків
       print(f' {row["Ім'я"]} - {row["Успішність"]}', end='')
       print(f' і він народився в {row["Рік народження"]}
                                                        року.')
        count += 1
    print(f'Усього у файлі {count + 1} рядків.')
```

Звертатися до елементів за назвою стовпця зручніше, крім того, це спрощує розуміння коду.

Зверніть увагу, що цикл for при першій ітерації буде записаний в row не шапка таблиці, а перший її рядок. Тому за виведення кількості рядків змінну count збільшили на 1.

Додаткові параметри об'єкта DictReader

`DictReader` має параметри:

- `dialect` - Набір параметрів форматування інформації. Докладніше про них нижче.
- `line_num` - Встановлює кількість рядків, які можна прочитати.
- `Fieldnames`- Визначає заголовки для стовпців, якщо не визначити атрибут, в нього запишуться елементи з першого прочитаного рядка файлу. Заголовки потрібні для того, щоб легко було зрозуміти, яка інформація міститься або повинна міститись у стовпці.

Наприклад, якби в classmates.csv не було першого рядка із заголовками, то можна було б його відкрити наступним чином:

```
fieldnames = ['Ім'я', 'Успішність', 'Рік народження']
file_reader = csv.DictReader(r_file,
                             fieldnames = fieldnames)
```

Також можна використовувати метод `__next__()` для отримання наступного рядка. Цей метод робить об'єкт reader ітерованим. Тобто він

викликається за кожної ітерації і повертає наступний рядок. Цей метод і використовується при кожній ітерації в циклі для отримання чергового рядка.

Запис до файлів

Для запису інформації в CSV файл необхідно створити об'єкт writer:

```
file_writer = csv.writer(w_file, delimiter = "\t")
```

Для запису файл даних використовується метод writerow(), який має наступний синтаксис:

```
writerow(["Ім'я", "Прізвище", "По батькові"])
```

Код програми для запису в CSV файл виглядає так:

```
import csv
with open("classmates.csv", mode="w", encoding='utf-8')
                                                  as w_file:
 file_writer = csv.writer(w_file, delimiter = ",",
                                      lineterminator="\r")
 file_writer.writerow(["Ім'я", "Клас", "Вік"])
 file_writer.writerow(["Женя", "3", "10"]))
 file_writer.writerow(["Саша", "5", "12"]))
 file_writer.writerow(["Маша", "11", "18"])
```

Зверніть увагу, що при записі використовувався lineterminator="r". Це роздільник між рядками таблиці, за умовчанням він \r\n.

Після виконання програми у файлі CSV буде наступний текст:

```
Ім'я,Клас,Вік
Женя,3,10
Саша, 5,12
Маша, 11,18
```

Як параметр метод writerow() приймає список, елементи якого будуть записані в рядок через символ-розділювач.

Запис у файл також можна здійснити за допомогою об'єкта `DictWriter`**.**

Важливо пам'ятати, що він потребує явної вказівки параметра fieldnames. Як аргумент методу writerow використовується словник.

Код програми виглядає так:

```
import csv
with open("classmates.csv", mode="w", encoding='utf-8') as w_file:

 names = ["Ім'я", "Вік"]
 file_writer = csv.DictWriter(w_file, delimiter = ",",
 lineterminator="\r",
 fieldnames=names)
 file_writer.writeheader()
 file_writer.writerow({"Ім'я": "Саша", "Вік": "6"})
 file_writer.writerow({"Ім'я": "Маша", "Вік": "15"})
 file_writer.writerow({"Ім'я": "Вова", "Вік": "14"})
```

Виведення у файл буде наступним:

```
Ім'я,Вік
Саша,6
Маша,15
Вова,14
```

Додаткові параметри `DictWriter`

Об'єкт writer також має атрибут dialect, який визначає, як форматуватимуться дані під час запису файл, про нього буде описано нижче.

Крім того, writer має методи:

- `writerows(rows)` – Записує всі елементи рядків.
- `writeheader()` –Виводить заголовки для стовпців. Заголовки повинні бути передані об'єкту writer у вигляді списку, як атрибут fieldnames. writeheader був використаний у попередньому прикладі.

Розглянемо застосування writerows:

```
file_writer.writerows([{"Ім'я": "Саша", "Вік": "6"},
 {"Ім'я": "Маша", "Вік": "15"},
 {"Ім'я": "Вова", "Вік": "14"}])
```

Діалекти

Щоб не вказувати формат вхідних і вихідних даних, певні параметри форматування згруповані в діалекти (`dialect`).

При створенні об'єкта reader або writer програміст може вказати потрібний діалект, крім того, деякі параметри діалекту можна перевизначити вручну, також вказавши їх при створенні об'єкта.

Для створення діалекту використовується команда:

```
register_dialect("ім'я", delimiter = "\t", ...)
```

Клас `Dialect` дозволяє визначити наступні атрибути форматування:

Атрибут	Значення
`delimiter`	Встановлює символ, за допомогою якого елементи розділяються у файлі. За замовчуванням використовується кома.
`doublequote`	Якщо `True`, то символ quotechar подвоюється, якщо `False`, то до символу qutechar додається ecsapechar як префікс.
`escapechar`	Рядок з одного символу, який використовується для екранування символу-розділювача.
`lineterminator`	Визначає роздільник для рядків, за замовчуванням використовується \r\n
`quotechar`	Визначає символ, який використовується для оточення символу-розділювача. За замовчуванням використовуються подвійні лапки, тобто `quotechar = '''`.
`quoting`	Визначає символ, який використовується для екранування роздільного символу (якщо не використовуються лапки).
`skipinitialspace`	Якщо встановити значення цього параметра в True, всі прогалини після символу-розділювача будуть ігноруватися.
`strict`	Якщо встановити в True, то при неправильному введенні CSV збуджуватиметься виняток Error.

Приклад використання

```
import csv
csv.register_dialect('my_dialect', delimiter=':',
                                              lineterminator="\r")
with open("classmates.csv", mode="w", encoding='utf-8') as w_file:
  file_writer = csv.writer(w_file, 'my_dialect')
  file_writer.writerow(["Ім'я", "Клас", "Вік"])
  file_writer.writerow(["Женя", "3", "10"]))
  file_writer.writerow(["Саша", "5", "12"]))
  file_writer.writerow(["Маша", "11", "18"])
```

В результаті отримаємо:

```
Ім'я:Клас:Вік
Женя:3:10
Саша:5:12
Маша:11:18
```

Приклад

```
import csv

FILENAME = "users.csv"

users = [
 [1, "математика", 75],
 [2, "програмування", 85],
 [3, "мат. аналіз", 34]
]

with open(FILENAME, "w", newline="") as file:
 writer = csv.writer(file)
 writer.writerows(users)

with open(FILENAME, "a", newline="") as file:
 user = [4, "веб прог", 80]
 writer = csv.writer(file)
 writer.writerow(user)

d={}
with open(FILENAME, "r", newline="") as file:
 reader = csv.reader(file)
 for row in reader:
 print(row[0], "-", row[1], "-", row[2])
 d[row[1]]=int(row[2])
print("d=",d)

#сформуємо список D1 зі словника та d
# та його сортуємо за ключем словника key=lambda x:x[0]
# якщо хочемо за значенням, пишіть key=lambda x:x[1]
# якщо хочемо у зворотному порядку, то пишіть
# key=lambda x:-x[1]
D1=sorted(d.items(),key=lambda x:-x[1]) # за ключами
```

```
print("D1=",D1) # друк списку

# відсортований список у словник D2 та його друк
D2={D1[i][0] : D1[i][1] for i in range(len(D1)) }
#друк таблицею D2
for kk, vv in D2.items():
 print(vv,'\t',kk)

ll=list(D2.items())
kkl = len (ll)-1
print("maximum score of", ll[0][1], "in the subject", ll[0][0])
print("minimum score of", ll[kkl][1],"in the subject", ll[kkl][0])

user1=[]
nn=0
for kk, vv in D2.items():
 user1.append([nn,kk,vv])
 nn+=1

namerow=["№", "предмет", "оцінка"]
with open(FILENAME, "w", newline="") as file: #encoding='utf-8'
 writer=csv.writer(file, delimiter=",", lineterminator="\n")
 writer.writerow(namerow)
 writer.writerows(user1)
```

Питання для самоконтролю до теми 8

1. Який формат файлів CSV?
2. Яка офісна програма може працювати із файлами формату CSV?
3. Як прочитати дані з файлу CSV?
4. Як записати дані у файл формату CSV?
5. Що таке діалекти параметрів формату читання/запису інформації файлів CSV?

Завдання до теми 8

1) Створити текстовий файл. Рядки файлу повинні містити порядковий номер, назву дисципліни та отриманий бал за нею. 2) Розробити програму, яка на основі цього файлу виводить на консоль таблицю. Таблиця повинна містити два стовпці – найменування предмета та кількість набраних балів. Дані у таблиці мають бути впорядковані за значеннями балів. 3) Вивести назву предметів, для яких отримано мінімальний та максимальний бал. Розрахувати середнє значення балу. 4) Програма повинна записати дані з отриманої таблиці у файл формату CSV з обов'язковим рядком з назвою стовпців.

9. МОДУЛЬ `shelve` (*.INI)

Для роботи з бінарними файлами Python може застосовуватися ще один модуль – **`shelve`**. Він зберігає об'єкти у файл із певним ключем. Потім за цим ключем може витягти збережений об'єкт з файлу. Процес роботи з даними через модуль `shelve` нагадує роботу зі словниками, які також використовують ключі для збереження та вилучення об'єктів.

Для відкриття файлу модуль shelve використовує функцію open():

```
open(шлях_до_файлу[, flag="c"[, protocol=None[,
                                    writeback=False]]])
```

Параметр `flag` може приймати значення:

- **c**: файл відкривається для читання та запису (значення за замовчанням). Якщо файл не існує, він створюється.
- **r**: файл відкривається лише для читання.
- **w**: файл відкривається для запису
- **n**: файл відкривається для запису Якщо файл не існує, він створюється. Якщо він існує, то він перезаписується

Для закриття підключення до файлу викликається метод close():

```
import shelve
d = shelve.open(filename)
d.close()
```

Або можна відкривати файл за допомогою оператора with.

Збережемо та рахуємо у файл кілька об'єктів:

```
import shelve

FILENAME = "states2"
With shelve.open(FILENAME) as states:
  states["London"] = "Great Britain"
  states["Paris"] = "France"
  states["Berlin"] = "Німеччина"
  states["Madrid"] = "Spain"

with shelve.open(FILENAME) as states:
  print(states["London"])
  print(states["Madrid"])
```

Запис даних передбачає встановлення значення для певного ключа:

```
states["London"] = "Great Britain"
```

А читання з файлу еквівалентне отриманню значення за ключом:

```
print(states["London"])
```

Як ключі використовуються рядкові значення.

При читанні даних, якщо запитуваний ключ відсутній, то генерується виняток. У цьому випадку перед отриманням ми можемо перевіряти наявність ключа за допомогою оператора in:

```
with shelve.open(FILENAME) as states:
  key = "Brussels"
  if key in states:
     print(states[key])
```

Можна також використовувати метод `get()`. Перший параметр методу - ключ, яким слід отримати значення, а другий – значення за замовчанням, яке повертається, якщо ключ не знайдено.

```
with shelve.open(FILENAME) as states:
  state = states.get("Brussels", "Undefined")
  print(state)
```

Використовуючи цикл for, можна перебрати всі значення файлу:

```
with shelve.open(FILENAME) as states:
for key in states:
   print(key," - ", states[key])
```

Метод keys() повертає всі ключі з файлу, а метод values() – всі значення:

```
with shelve.open(FILENAME) as states:
for city in states.keys():
  print(city, end=" ") # London Paris Berlin Madrid
print()
for country in states.values():
  print(country, end=" ") # Great Britain
                          # France Germany Spain
print()
```

Ще один метод `items()` повертає набір кортежів. Кожен кортеж містить ключ та значення.

```
with shelve.open(FILENAME) as states:
  for state in states.items():
     print(state)
```

Консольне виведення:

```
("London", "Great Britain")
("Paris", "France")
("Berlin", "Німеччина")
("Madrid", "Spain")
```

Оновлення даних

Для зміни даних достатньо присвоїти по ключу нове значення, а для додавання даних визначити новий ключ:

```
import shelve
FILENAME = "states2"
with shelve.open(FILENAME) as states:
  states["London"] = "Great Britain"
  states["Paris"] = "France"
  states["Berlin"] = "Німеччина"
```

```
  states["Madrid"] = "Spain"

with shelve.open(FILENAME) as states:
  states["London"] = "United Kingdom"
  states["Brussels"] = "Belgium"
  for key in states:
    print(key, "-", states[key])
```

Видалення даних

Для видалення з одночасним отриманням можна використовувати функцію pop(), в яку передається ключ елемента та значення за замовчанням, якщо ключ не знайдено:

```
with shelve.open(FILENAME) as states:
  state = states.pop("London", "NotFound")
  print(state)
```

Також для видалення може застосовуватись оператор del:

```
with shelve.open(FILENAME) as states:
  del states["Madrid"] # видаляємо об'єкт з ключем Madrid
```

Для видалення всіх елементів можна використовувати метод clear():

```
with shelve.open(FILENAME) as states:
  states.clear()
```

Питання для самоконтролю до теми 9

1. Яке призначення модуля shelve?
2. Який формат файлів використовується у модулі shelve?
3. Які методи (функції) входять до складу модуля shelve?
4. Як записати у файл запису формату КЛЮЧ=ЗНАЧЕННЯ?
5. Як прочитати з файлу значення заданого ключа?

Завдання до теми 9

Створити текстовий файл. Рядки файлу повинні містити порядковий номер, коротку назву дисципліни та отриманий бал.

Розробити програму, яка на основі цього файлу створить файл формату `.ini`. Ключем вважати назву дисципліни, значенням – бал.

Програма повинна виводити таблицю, що містить стовпці «номер по порядку», «дисципліна» та «бал». Запитавши у користувача номер дисципліни та нове значення балу, виконати зміни рядка таблиця. Після завершення роботи – зберегти змінену таблицю у файл формату `.ini` та вивести рядки таблиці на консоль.

10. МОДУЛЬ os – РОБОТА З ФАЙЛОВОЮ СИСТЕМОЮ

Ряд можливостей роботи з каталогами і файлами надає вбудований модуль os. Він містить багато функцій, нижче розглянуті лише основні з них:

- `mkdir()`: створює нову папку
- `rmdir()`: видаляє папку
- `rename()`: перейменовує файл
- `remove()`: видаляє файл

Створення та видалення папки

Для створення папки застосовується функція mkdir(), в яку передається шлях до створюваної папки:

```
import os

    шлях щодо поточного скрипту
os.mkdir("hello")
    абсолютний шлях
os.mkdir("c://somedir")
os.mkdir("c://somedir/hello")
```

Для видалення папки використовується функція rmdir(), в яку передається шлях до папки, що видаляється:

```
import os
    шлях щодо поточного скрипту
os.rmdir("hello")
    абсолютний шлях
os.rmdir("c://somedir/hello")
```

Перейменування файлу

Для перейменування викликається функція `rename(source, target)`, перший параметр якої шлях до вихідного файлу, а другий - нове ім'я файлу. Як шляхи можуть використовуватися як абсолютні, так і відносні.

Наприклад, нехай у папці C://SomeDir/ розміщується файл somefile.txt. Перейменуємо його на файл "hello.txt":

```
import os
os.rename("C://SomeDir/somefile.txt",
                                "C://SomeDir/hello.txt")
```

Видалення файлу

Для видалення викликається функція remove(), яку передається шлях до файлу:

```
importos

os.remove("C://SomeDir/hello.txt")
```

Існування файлу

Якщо спробувати відкрити файл, який немає, то Python «викине» виняток FileNotFoundError.

Для вилову виключення можна використовувати конструкцію try...except. Однак можна вже до відкриття файлу перевірити, чи існує він за допомогою методу `os.path.exists(path)`.

У цей метод передається повне ім'я файлу (шлях та ім'я), існування якого необхідно перевірити:

```
filename = input("Введіть шлях до файлу: ")
if os.path.exists(filename):
  print("Вказаний файл існує")
else:
  print("Файл не існує")
```

Робота з операційною системою

Модуль `os` також надає безліч функцій для роботи з операційною системою, причому їхня поведінка, як правило, не залежить від ОС, тому програми залишаються кросплатформними. Нижче наведені найчастіше використовувані з них (деякі функції цього модуля підтримуються не всіма ОС).

`os.name` – ім'я операційної системи. Доступні варіанти: `'posix', 'nt', 'mac', 'os2', 'ce', 'java'`.

`os.environ` – словник змінних оточення. Змінюваний (можна додавати та видаляти змінні оточення).

`os.getlogin()` – Ім'я користувача, що увійшов до терміналу (Unix).

`os.getpid()` – поточний `id` процесу.

`os.uname()` – інформація про ОС. повертає об'єкт з атрибутами: sysname – ім'я операційної системи, nodename – ім'я машини в мережі (визначається реалізацією), release – реліз, `version` – версія, `machine` – ідентифікатор машини.

`os.access(path, mode, *, dir_fd=None, effective_ids=False, follow_symlinks=True)` – перевірка доступу до об'єкта для поточного користувача.

Прапори:

`os.F_OK` – об'єкт існує, `os.R_OK` – доступний на читання,

`os.W_OK` – доступний на запис, `os.X_OK` – доступний на виконання.

`os.chdir(path)` – зміна поточної директорії.

`os.chmod(path, mode, *, dir_fd=None, follow_symlinks=True)` – зміна прав доступу до об'єкта (`mode` – вісімкове число).

`os.chown(path, uid, gid, *, dir_fd=None, follow_symlinks=True)` – змінює `id` власника та групи (Unix).

`os.getcwd()` – поточна робоча директорія.

`os.link(src, dst, *, src_dir_fd=None, dst_dir_fd=None, follow_symlinks=True)` – створює жорстке посилання.

`os.listdir(path=".")` – список файлів та директорій у папці.

`os.mkdir(Path, mode = 0x777, *, dir_fd = None)` – створює директорію. OSError, якщо директорія існує.

`os.makedirs(Path, mode = 0x777, exist_ok = False)` – створює директорію, створюючи при цьому проміжні директорії.

`os.remove(path, *, dir_fd=None)` – видаляє шлях до файлу.

`os.rename(src, dst, *, src_dir_fd=None, dst_dir_fd=None)` – перейменовує файл або директорію з `src` на `dst`.

`os.renames(old, new)` – перейменовує `old` на `new`, створюючи проміжні директорії.

`os.replace(src, dst, *, src_dir_fd=None, dst_dir_fd=None)` – перейменовує з `src` на `dst` з примусовою заміною.

`os.rmdir(Path, *, dir_fd = None)` – видаляє порожню директорію.

`os.removedirs(path)` – видаляє директорію, потім намагається видалити батьківські директорії, і видаляє їх рекурсивно, поки вони порожні.

`os.symlink(source, link_name, target_is_directory = False, *, dir_fd = None)` – створює символічне посилання на об'єкт.

`os.sync()` – записує всі дані на диск (Unix).

`os.truncate(path, length)` – обрізає файл до довжини `length`.

`os.utime(path, times=None, *, ns=None, dir_fd=None, follow_symlinks=True)` – модифікація часу останнього доступу та зміни файлу. Або `times` – кортеж (час доступу в секундах, час зміни в секундах), або `ns` – кортеж (час доступу в наносекундах, час зміни в наносекундах).

`os.walk(top, topdown = True, on error = None, followlinks = False)` – генерація імен файлів у дереві каталогів, зверху вниз (якщо topdown дорівнює True), або знизу вгору (якщо False). Для кожного каталогу функція walk повертає кортеж (шлях до каталогу, список каталогів, список файлів).

Приклад – вивести на консоль список усіх *.txt файлів, розміщених у папці l:\dist та її підпапках:

```
import os
from os.path import join
for (root, dirs, files) in os.walk('l:\\dist'):
      for filename in files:
            if filename.endswith('.txt'):
                  thefile = os. path.join(root, filename)
                  print( os.path.getsize(thefile), thefile)
```

`os.system(command)` – виконує системну команду, повертає код її завершення (у разі успіху 0).

`os.urandom(n)` – `n` випадкових байт. Можливе використання цієї функції у криптографічних цілях.

Питання для самоконтролю до теми 10

1. Основне призначення функцій OS?
2. Як використовувати модуль OS для створення папки?
3. Як використовувати модуль OS перейменувати файл?
4. Як використовувати модуль OS для зміни поточної папки?
5. Як використовуючи модуль OS "обійти" всі папки, зареєстровані в заданій?

Завдання до теми 10

1. Розробити програму (скрипт) мовою Python, з використанням відповідних пакетів, для отримання статистичних даних про файли та папки, що знаходяться на зовнішніх носіях.

2. Модифікувати «допоміжний скрипт» (див. нижче) до виконання тестування розробленої програми.

Вимоги до програми:

– отримати статистичні дані про файли та підпапки, розташовані в заданій папці;

– в отримані дані включити сумарну інформацію з усіх файлів і підпапкам заданої папки.

Результат роботи програми оформити у вигляді двох таблиць:

Таблиця №1:

Найменування папки (підпапки), кількість підпапок у цій папці та їх сумарний розмір, кількість файлів у цій папці та їх сумарний розмір.

Таблиця № 2:

Ім'я типу файлу та сумарний розмір; % від усієї кількості всіх файлів цього типу, виявлених у всіх заданих папках та підпапках.

Вихідні дані до роботи необхідно отримувати, використовуючи параметри командного рядка.

В обов'язковому порядку передбачити виведення опису правил використання програми на консоль, використовуючи опцію `/?` або – `help`.

Для перевірки працездатності програми рекомендується побудувати набір тестової структури досліджуваного дерева каталогів та файлів у них.

Для цього можна, наприклад, скористатися таким скриптом:

```
import pathlib, itertools, glob, shutil
import random
# root dir
path = 'D://tx//'
# type file & max size (Kb)
ftype = {'.py' : 10, '.txt' : 100, '.xyz' : 5, '.bak' : 1 }
# sub dir name
tdir=  {'script', 'text', 'file', 'ddd' }
# combinations of name of file
combin=[1, 2, 3, 4, 5]
for d in tdir:
    comb = itertools.combinations(combin, r=2)
```

```
for a, b in comb:
    pathlib.Path(path, d).mkdir(parents=True, exist_ok=True)
    for ext, size in ftype.items():
        name = f'{d}{a}{b}{ext}'
        pathlib.Path(path, d, name).touch(exist_ok=True)
        full_name=pathlib.Path(path, d, name)
        f=open(full_name,"wt")
        n=size*int(1024*(random.random()+1))
        buf=n*ext[1:]
        f.write(buf)
        f.close()
```

11. ПРИКЛАД КОМАНД РОБОТИ З ФАЙЛАМИ ТА ФАЙЛОВОЮ СИСТЕМОЮ

Розглянемо 8 вкрай важливих команд для роботи з файлами, папками і файловою системою в цілому.

Показати поточний каталог

Найпростіша і водночас одна з найважливіших команд для Python-розробника. Вона потрібна тому, що найчастіше розробники мають справу із відносними шляхами. Але в деяких випадках важливо знати, де ми знаходимося.

Відносний шлях хороший тим, що працює для всіх користувачів з будь-якими системами, кількістю дисків тощо.

Щоб показати поточний каталог використовуємо вбудовану в Python OS-бібліотеку:

```
import os
os.getcwd()
```

Шлях, що повертається, є абсолютним.

Перевіряємо, чи існує файл або каталог

Перш ніж задіяти команду створення файлу або каталогу, варто переконатися, що аналогічних елементів немає. Це допоможе уникнути ряду помилок під час роботи програми, включаючи перезапис існуючих елементів з даними.

Функція `os.path.exists ()` приймає аргумент рядкового типу, який може бути або ім'ям каталогу або файлом.

Перевіримо, чи існує каталог `sample_data`:

```
os.path.exists('sample_data')
```

```
[3]  os.path.exists('sample_data')

     True
```

Ця ж команда підходить і для роботи з файлами:

```
os.path.exists('sample_data/README.md')
```

```
[4]  os.path.exists('sample_data/README.md')

     True
```

Якщо папки чи файлу немає, команда повертає `false`.

```
[5]  os.path.exists('foobar')

     False
```

Об'єднання компонентів шляху

У попередньому прикладі використовувався символ "/" для роздільника компонентів шляху. У принципі, це нормально, але не рекомендується. Якщо необхідно, щоб програма була кросплатформною, такий варіант не підходить. Деякі старі версії ОС Windows розпізнають лише символ "\" як роздільник.

Python вирішує цю проблему завдяки функції `os.path.join()`.

Давайте перепишемо варіант із прикладу в попередньому пункті, використовуючи цю функцію:

```
os.path.exists(os.path.join('sample_data', 'README.md'))
```

```
[7]  os.path.exists(os.path.join('sample_data', 'README.md'))

     True
```

Створення директорії

Створимо директорію з ім'ям `test_dir` усередині поточної директорії. Для цього можна використовувати функцію `os.mkdir()`:

```
os.mkdir('test_dir')
```

Погляньмо, як це працює на практиці.

```
[9]  print(f"test_dir existing: {os.path.exists('test_dir')}")
     os.mkdir('test_dir')
     print(f"test_dir existing: {os.path.exists('test_dir')}")

     test_dir existing: False
     test_dir existing: True
```

Якщо ми спробуємо створити каталог, який вже існує, то отримаємо виняток:

```
[11] os.mkdir('test_dir')

---------------------------------------------------------------------------
FileExistsError                           Traceback (most recent call last)
<ipython-input-11-48d7b58469aa> in <module>()
----> 1 os.mkdir('test_dir')

FileExistsError: [Errno 17] File exists: 'test_dir'
```

Саме тому рекомендується завжди перевіряти наявність каталогу з певною назвою перед створенням нового:

```
if not os.path.exists('test_dir'):
   os.mkdir('test_dir')
```

Ще одна порада щодо створення каталогів. Іноді нам потрібно створити підкаталоги з рівнем вкладеності 2 або більше. Якщо ми досі використовуємо `os.mkdir()`, нам потрібно буде зробити це кілька разів.

І тут ми можемо використовувати `os.makedirs()`. Ця функція створить усі проміжні каталоги:

```
os.makedirs(os.path.join('test_dir', 'level_1',
                                  'level_2', 'level_3'))
```

Ось що виходить у результаті.



Показ вмісту директорії

Ще одна корисна команда – `os.listdir()`. Вона показує весь вміст каталогу.

Команда відрізняється від `os.walk()`, де остання рекурсивно показує все, що знаходиться в каталозі. `os.listdir()` набагато простіше у використанні, тому що просто повертає список вмісту:

```
os.listdir('sample_data')
```

```
[13] os.listdir('sample_data')

     ['README.md',
      'anscombe.json',
      'california_housing_train.csv',
      'california_housing_test.csv',
      'mnist_train_small.csv',
      'mnist_test.csv']
```

У деяких випадках потрібно щось більш просунуте, наприклад, пошук усіх CSV файлів у каталозі «sample_data». У цьому випадку найпростіший спосіб – використовувати вбудовану бібліотеку `glob` (див. нижче):

```
from glob import glob
list(glob(os.path.join('sample_data', '*.csv'))))
```

```
[14] from glob import glob
```

```
[15] list(glob(os.path.join('sample_data', '*.csv')))
```

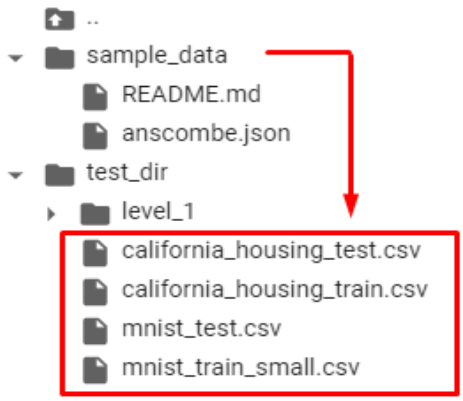
```
['sample_data/california_housing_train.csv',
 'sample_data/california_housing_test.csv',
 'sample_data/mnist_train_small.csv',
 'sample_data/mnist_test.csv']
```

Переміщення файлів

Якщо необхідно перемістити файли з однієї папки до іншої, то рекомендований спосіб – ще одна вбудована бібліотека `shutil`.

Перемістимо всі CSV-файли з директорії `sample_data` до директорії `test_dir`:

```
import shutil
for file in list(glob(os.path.join('sample_data',
                                   '*.csv')))):
    shutil.move(file, 'test_dir')
```



До речі, є два способи виконати задумане. Наприклад, ми можемо використовувати бібліотеку OS, якщо не хочеться імпортувати додаткові бібліотеки. Як `os.rename`, так і `os.replace` підходять вирішення завдання.

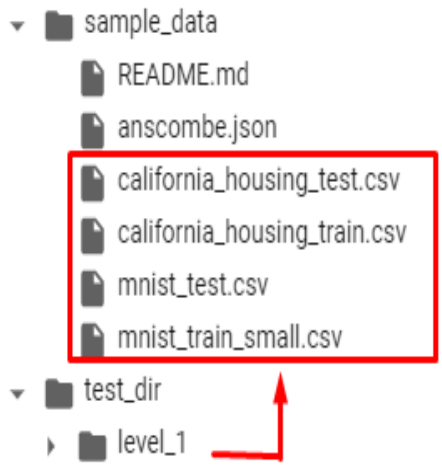
Але обидві вони недостатньо «розумні», щоб дозволити перемістити файли в каталог.

```
[21] for file in list(glob(os.path.join('test_dir', '*.csv'))):
         os.rename(file, 'sample_data')
```

```
---------------------------------------------------------------------------
IsADirectoryError                         Traceback (most recent call last)
<ipython-input-21-9675c94bbcdd> in <module>()
      1 for file in list(glob(os.path.join('test_dir', '*.csv'))):
----> 2     os.rename(file, 'sample_data')

IsADirectoryError: [Errno 21] Is a directory: 'test_dir/california_housing_train.csv' -> 'sample_data'
```

Щоб усе це працювало, потрібно явно вказати ім'я файлу на місці призначення. Нижче код, який це дозволяє зробити:

```
for file in list(glob(os.path.join('test_dir',
                                   '*.csv'))):
   os.rename(
             file,
             os.path.join(
               'sample_data',
                 os.path.basename(file)
             ))
```



Тут функція `os.path.basename()` призначена для вилучення імені файлу зі шляху з будь-якою кількістю компонентів.

Інша функція, `os.replace()`, робить те саме. Але різниця в тому, що `os.replace()` не залежить від платформи, тоді як `os.rename()` працюватиме лише в системі Unix/Linux.

Ще один мінус у тому, що обидві функції не підтримують переміщення файлів з різних файлових систем, на відміну від `shutil`.

Тому рекомендується використовувати `shutil.move()` для переміщення файлів.

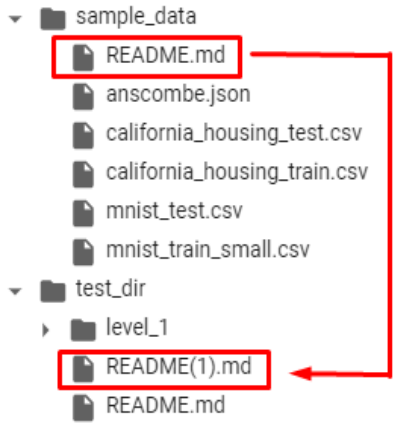
Копіювання файлів

Аналогічно `shutil` підходить і для копіювання файлів з вже згаданих причин.

Якщо потрібно скопіювати файл README.md з папки `sample_data` в папку `test_dir`, допоможе функція `shutil.copy()`:

```
shutil.copy(
   os.path.join('sample_data', 'README.md'),
   os.path.join('test_dir')
             )
```

```
[26] shutil.copy(
         os.path.join('sample_data', 'README.md'),
         os.path.join('test_dir', 'README(1).md')
     )

     'test_dir/README(1).md'
```



Видалення файлів та папок

Коли потрібно видалити файл, потрібно скористатися командою os.remove():

```
os.remove(os.path.join('test_dir', 'README(1).md'))
```

Якщо потрібно видалити каталог, використовуємо `os.rmdir()`:

```
os.rmdir(os.path.join('test_dir', 'level_1', 'level_2',
'level_3'))
```



Однак `os.rmdir()` може видалити лише порожній каталог.

На наведеному рисунку бачимо, що можна видалити лише каталог level_3.

Що, якщо необхідно рекурсивно видалити каталог level_1? У цьому випадку використовуємо `shutil`.

```
[28] os.rmdir(os.path.join('test_dir', 'level_1'))

---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)
<ipython-input-28-352a126b9aab> in <module>()
----> 1 os.rmdir(os.path.join('test_dir', 'level_1'))

OSError: [Errno 39] Directory not empty: 'test_dir/level_1'
```

Функція `shutil.rmtree()` зробить все, що потрібно:

```
shutil.rmtree(os.path.join('test_dir', 'level_1'))
```

Користуватися нею потрібно обережно, оскільки вона безповоротно видаляє весь вміст каталогу.

Питання для самоконтролю до теми 11

1. Як перевірити існування файлу чи каталогу?
2. Як правильно поєднати компоненти шляху до файлу незалежно від формату ОС?
3. Як створити директорію (папку) та набір папок у заданій директорії?
4. Як перемістити або скопіювати групу файлів та задану папку?
5. Як видалити об'єкти файлової системи?

Завдання до теми 11

Напишіть скрипт, який створює систему каталогів, імена яких задаються списком. Елементом списку може бути ім'я створюваного каталогу або список з ім'ям створюваного каталогу та імен підкаталогів, що створюються в ньому.

Наприклад, список

```
['d:\temp', ' x:\abc', ['z:\xyz', 'abc', 'z1', 'z2'], 'q:\123' ]
```

Повинен створити каталоги

```
d:\temp,
x:\abc,
z:\xyz,
z:\xyz\abc,
z:\xyz\z1,
z:\xyz\z2,
q:\123
```

12. МОДУЛЬ `shutil`

Модуль `shutil` містить набір функцій високого рівня обробки файлів, груп файлів, і папок. Функції модуля дозволяють копіювати, переміщати та видаляти файли та папки. Часто використовується разом із модулем os.

Операції над файлами та директоріями

`shutil.copyfileobj(fsrc, fdst [, length])`

Скопіювати вміст одного файлового об'єкта (`fsrc`) в інший (`fdst`).

Необов'язковий параметр `length` – розмір буфера при копіюванні (щоб весь, можливо величезний, файл не читався повністю).

При цьому, якщо позиція покажчика `fsrc` не 0 (тобто до цього було зроблено щось на зразок `fsrc.read(47))`, то буде копіюватися вміст починаючи з поточної позиції, а не з початку файлу.

`shutil.copyfile(src, dst, follow_symlinks = True)`

Копіює вміст (але не метадані) файлу `src` у файл `dst`.

Повертає `dst` (тобто куди файл було скопійовано). `src` та `dst` це рядки – шляхи до файлів. `dst` має бути повним ім'ям файлу.

Якщо `src` і `dst` є один і той же файл, виключення `shutil.SameFileError`.

Якщо `dst` існує, він буде перезаписаний.

Якщо `follow_symlinks=False` та `src` є посиланням на файл, то буде створено нове символічне посилання замість копіювання файлу, на який це символічне посилання вказує.

`shutil.copymode(src, dst, follow_symlinks = True)`

копіює права доступу з `src` в `dst`. Вміст файлу, власник і група не змінюються.

`shutil.copystat(src, dst, follow_symlinks = True)`

копіює права доступу, час останнього доступу, останньої зміни та прапори `src` в `dst`. Вміст файлу, власник і група не змінюються.

`shutil.copy(src, dst, follow_symlinks=True)`

копіює вміст файлу `src` у файл чи папку `dst`. Якщо `dst` є директорією, файл буде скопійовано з тією самою назвою, що була у `src`. Функція повертає шлях до розташування нового скопійованого файлу.

Якщо `follow_symlinks=False` і `src` це посилання, `dst` буде посиланням.

Якщо `follow_symlinks=True`, і src це посилання, `dst` буде копією файлу, який посилається `src. copy()` копіює вміст файлу та права доступу.

`shutil.copy2(src, dst, follow_symlinks = True)`

як `copy()`, але намагається копіювати всі метадані.

```
shutil.copytree(src, dst, symlinks=False, ignore=None,
          copy_function=copy2, ignore_dangling_symlinks=False)
```

Рекурсивно копіює все дерево директорій з коренем у `src`, повертає директорію призначення.

Директорія `dst` має існувати. Вона буде створена разом із пропущеними батьківськими директоріями.

Права та час у директорій копіюються `copystat()`, файли копіюються за допомогою функції `copy_function` (за замовчуванням `shutil.copy2()`).

Якщо `symlinks=True`, посилання в дереві `src` будуть посиланнями в `dst` і метадані будуть скопійовані настільки, наскільки це можливо.

Якщо `symlinks=False` (за замовчанням), буде скопійовано вміст і метадані файлів, на які вказували посилання.

Якщо `symlinks=False`, якщо файлу, на який вказує посилання, немає, буде додано виняток до списку помилок, у виключенні `shutil.Error` в кінці копіювання.

Можна встановити прапорець `ignore_dangling_symlinks=True`, щоб приховати цю помилку.

Якщо `ignore` не `None`, то це має бути функція, яка приймає як аргументи ім'я директорії, в якій зараз `copytree()`, і список вмісту, що повертається `os.listdir()`. Враховуючи, що `copytree()` викликається рекурсивно, `ignore` викликається 1 раз для кожної піддиректорії. Вона повинна повертати список об'єктів щодо поточного імені директорії (тобто підмножина елементів у другому аргументі). Ці об'єкти не будуть скопійовані.

`shutil.ignore_patterns(*patterns)`

функція, яка створює функцію, яка може бути використана як `ignore` для `copytree()`, ігноруючи файли та директорії, які відповідають glob-стилі шаблонів.

Наприклад:

```
copytree(source, destination,
             ignore=ignore_patterns('*.pyc', 'tmp*'))
```

Копіює всі файли, крім тих, що закінчуються на .pyc або починаються з `tmp`

`shutil.rmtree(path, ignore_errors=False, onerror=None)`

видаляє поточну директорію та всі піддиректорії; `path` повинен вказувати на директорію, а не символічне посилання.

Якщо `ignore_errors=True`, то помилки, що виникають в результаті невдалого видалення, будуть проігноровані. Якщо `False` (за замовчуванням), ці помилки будуть передаватися обробнику `onerror`, або, якщо його немає, то виняток.

На ОС, які підтримують функції на основі файлових дескрипторів, за замовчуванням використовується версія `rmtree()`, не вразлива до атак символічних посилань.

На інших платформах це не так: за вибраного часу та обставин "хакер" може, маніпулюючи посиланнями, видалити файли, які недоступні йому в інших обставинах.

Щоб перевірити, чи вразлива система до подібних атак, можна використовувати атрибут `rmtree.avoids_symlink_attacks`.

Якщо заданий `oneerror`, це має бути функція з 3 параметрами: `function, path, excinfo`.

Перший параметр, `function`, це функція, яка створила виняток; вона залежить від платформи та інтерпретатора. Другий параметр, `path`, це шлях, що передається функції. Третій параметр `excinfo` – це інформація про виключення, що повертається `sys.exc_info`(). Винятки, спричинені на небезпеку, не обробляються.

```
shutil.move(src, dst, copy_function = copy2)
```

рекурсивно переміщає файл або директорію (`src`) до іншого місця (`dst`), і повертає місце призначення.

Якщо `dst` – існуюча директорія, то src переміщається всередину директорії. Якщо `dst` існує, але з директорія, воно може бути перезаписано.

```
shutil.disk_usage(Path)
```

повертає статистику використання дискового простору з атрибутами `total`, `used` і `free`, в байтах.

```
shutil.chown(path, user=None, group=None)
```

змінює власника та/або групу у файлу чи директорії.

```
shutil.which(cmd, mode=os.F_OK | os.X_OK, path=None)
```

Повертає шлях до виконуваного файлу по заданій команді.

Якщо немає відповідності з жодним файлом, то `None`. `mode` це права доступу, потрібні від файлу, за замовчуванням шукає лише виконувані.

Архівація

Високорівневі функції для створення та читання архівованих та стислих файлів. Засновані на функціях із модулів `zipfile` та `tarfile`.

```
shutil.make_archive(base_name, format[, root_dir[,
base_dir[, verbose[, dry_run[, owner[, group[, logger]]]]]]])
```

Створює архів і повертає його ім'я.

`base_name` це ім'я файлу для створення, включаючи шлях, але не включаючи розширення (не потрібно писати "`.zip`" та інше).

`format` – формат архіву.

`root_dir` – директорія (щодо поточної), яку ми архівуємо.

`base_dir` – директорія, в яку архівуватиметься (тобто всі файли в архіві будуть у цій папці).

Якщо `dry_run=True`, архів не буде створений, але операції, які мали бути виконані, запишуться в `logger`.

`owner` та `group` використовуються при створенні tar-архіву.

`shutil.get_archive_formats()`

Список доступних форматів для архівування.

```
>>> shutil.get_archive_formats()
[('bztar', "bzip2'ed tar-file"),
('gztar', "gzip'ed tar-file"),
('tar', 'uncompressed tar file'),
('xztar', "xz'ed tar-file"),
('zip', 'ZIP file')]
```

`shutil.unpack_archive(filename[, extract_dir[, format]])`

Розпаковує архів `filename` – повний шлях до архіву.

`extract_dir` – те, куди витягуватиметься вміст (за замовчуванням у поточну).

`format` – формат архіву (за замовчуванням намагається вгадати розширення файлу).

`shutil.get_unpack_formats()` – список доступних форматів для розархівування.

Запит розміру терміналу виведення

`shutil.get_terminal_size(fallback = (columns, lines))`

Повертає розмір вікна терміналу.

`fallback` повернеться, якщо не вдалося дізнатися про розмір терміналу (термінал не підтримує такі запити, або програма працює без терміналу). За замовчуванням (80, 24).

```
>>> shutil.get_terminal_size()
os.terminal_size(columns=102, lines=29)
>>> shutil.get_terminal_size() # Зменшили вікно
os.terminal_size(columns=67, lines=17)
```

Питання для самоконтролю до темі 12

1. Яким є основне призначення програм модуля `shelve`?
2. Як виконати копіювання файлових об'єктів за допомогою модуля `shelve`?
3. Як виконати копіювання файлів із заданої папки та з усіх її під папок у вказану директорію використовуючи модуль `shelve`?
4. Як встановити шаблон для групи файлів, що копіюються, використовуючи модуль `shelve`?
5. Як працювати з архівами, використовуючи модуль `shelve`?

Завдання до теми 12.

Напишіть програму архівування всіх файлів із зазначеної папки та її під папок до архіву `zip` формату. Ім'я архіву та ім'я папки вказуються як параметри

командного рядка виклику програми. Програма повинна вивести на консоль кількість за архівованих файлів, їх сумарний розмір, розмір файлу архіву, що вийшов, і значення відсотка «архівації».

13. ПРИКЛАДИ ВИКОРИСТАННЯ МОДУЛЯ `SHUTIL`

Копіювання файлу

Функція `shutil.copyfile()` копіює вміст джерела в місце призначення та викликає виключення `IOError`, якщо у нього немає дозволу на запис до файлу призначення.

```
>>> import shutil, os
>>> from glob import glob
# створимо тимчасову директорію
>>> os.mkdir('example')
# створимо тестовий файл
>>> open('example/test_file.txt', 'w').close()
# Копіювання
>>> shutil.copyfile('example/test_file.txt',
'example/test_file.txt.copy')
# 'example/test-file.txt.copy'
# дивимося результат
>>> pprint.pprint(glob('example/*'))
# ['example/test_file.txt.copy', 'example/test_file.txt']
# видаляємо
>>> shutil.rmtree('example')
```

Функція `shutil.copy()` інтерпретує ім'я вихідного файлу як інструмент командного рядка. Якщо шлях призначення вказаний як каталог, а не файл, то в каталозі створюється новий файл із використанням його базового імені.

```
>>> import shutil, os
>>> from glob import glob
# створимо тестовий файл
>>> open('shutil_copy.txt', 'w').close()
# створимо тимчасову директорію
>>> os.mkdir('example')
>>> glob('example/*')
# []
# Копіюємо
>>> shutil.copy('shutil_copy.txt', 'example')
# 'example/shutil_copy.txt'
# дивимося результат
>>> glob('example/*')
# ['example/shutil_copy.txt']
# видаляємо
>>> shutil.rmtree('example')
```

Рекурсивне копіювання каталогу

Функція `shutil.copytree()` рекурсивно копіює весь каталог.

```
# для того, щоб протестувати функцію copytree()
# створимо ієрархію каталогів
import pathlib, itertools, glob, shutil
```

```
path = 'test/'
tmp = {'script':'.py', 'text':'.txt'}
for d, ext in tmp.items():
    comb = itertools.combinations([d, 1, 0], r=2)
    for a, b in comb:
        name = f'{a}{b}{ext}'
        pathlib.Path(path, d).mkdir(parents=True, exist_ok=True)
        pathlib.Path(path, d, name).touch(exist_ok=True)
f = glob.glob('test/**/*', recursive=True)
print(f)
# ['test/text', 'test/script', 'test/text/10.txt',
# 'test/text/text1.txt', 'test/text/text0.txt',
# 'test/script/10.py', 'test/script/script0.py',
# 'test/script/script1.py']
# копіюємо
shutil.copytree('test', 'test_cp')
# дивимося результат
f_cp = glob.glob('test_cp/**/*', recursive=True)
print(f_cp)
# ['test/text', 'test/script', 'test/text/10.txt',
# 'test/text/text1.txt', 'test/text/text0.txt',
# 'test/script/10.py', 'test/script/script0.py',
# 'test/script/script1.py']
# видаляємо
shutil.rmtree('test_cp')
shutil.rmtree('test')
```

Вибіркове рекурсивне копіювання файлів каталогу

Приклад, який використовує помічник `shutil.ignore_patterns()`.

Тут копіюється все, крім файлів .py і файлів або каталогів, ім'я яких починається з `tmp`.

```
від shutil import copytree, ignore_patterns
copytree(source, destination, ignore=ignore_patterns('*.py',
                                                      'tmp*'))
```

Інший приклад, який використовує аргумент `ignore` для додавання виклику логування копіювання файлів:

```
from shutil import copytree
import logging
def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored
copytree(source, destination, ignore=_logpath)
```

Рекурсивне видалення каталогу

Робота функції `shutil.rmtree()`,яка виконує рекурсивне видалення каталогу демонструвалася вище. Розберемо ситуації складніше.

У цьому прикладі показано, як видалити дерево каталогів у Windows, де для деяких файлів встановлений біт тільки для читання.

Він використовує зворотний виклик на `error`, щоб очистити біт `readonly` і повторити спробу видалення.

```
import os, stat
import shutil
def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)
shutil.rmtree(directory, onerror=remove_readonly)
```

У функції `shutil.rmtree()` так само можна використовувати помічник `shutil.ignore_patterns()`для вибіркового рекурсивного видалення файлів, як це робиться у вибірковому рекурсивному копіюванні файлів.

Приклад реалізації функції `shutil.copytree()`

Цей приклад є реалізацією функції `shutil.copytree()`з опущеним рядком документації. Він демонструє багато інших функцій, що надаються цим модулем.

```
def copytree(src, dst, symlinks=False):
    names = os.listdir(src)
    os.makedirs(dst)
    errors = []
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
                copy2(srcname, dstname)
            # XXX What about devices, sockets etc.?
        except OSError as why:
            errors.append((srcname, dstname, str(why)))
        # catch the Error from the recursive copytree
                                                so that we can
        # continue with other files
        except Error as err:
            errors.extend(err.args[0])
    try:
        copystat(src, dst)
    except OSError as why:
        # can't copy file access times on Windows
```

```
        if why.winerror is None:
            errors.extend((src, dst, str(why)))
    if errors:
        raise Error(errors)
```

Архівування каталогів

У цьому прикладі створимо архів tar-файлів з використанням `gzip`, що містить усі файли, знайдені в каталозі `.ssh` користувача:

```
from shutil import make_archive
import os
archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
make_archive(archive_name, 'gztar', root_dir)
```

Отриманий архів містить:

```
~$ tar -tzvf /home/docs-python/myarchive.tar.gz
drwx------ docs-python/docs-python 0 2019-12-17 16:57 ./
-rw------- docs-python/docs-python 1554 2019-12-17
16:53 ./known_hosts
-rw------- docs-python/docs-python 1554 2019-12-17
14:06 ./known_hosts.old
-rw------- docs-python/docs-python 1679 2019-09-16 12:02 ./id_rsa
-rw-r--r-- docs-python/docs-python 399 2019-09-16
12:02 ./id_rsa.pub
```

Питання для самоконтролю до теми 13

1. Як скопіювати файл?
2. Як рекурсивно виконати коригування файлів з каталогу та всіх його підкаталогів?
3. Як задати шаблон файлів, що копіюються?
4. Як рекурсивно видалити гілку каталогу?
5. Як створити архів заданої групи файлів?

Завдання до теми 13

Напишіть програму копіювання всіх файлів із зазначеної папки та її під папок до заданої папки. Ім'я папок вказуються як параметри командного рядка виклику програми. Програма повинна вивести на консоль кількість скопійованих файлів, їх сумарний розмір.

14. Модуль `pathlib`

Python 3 включає модуль pathlib для маніпуляції шляхами файлових систем незалежно від операційної системи.

`Pathlib` схожий на модуль `os.path`, але `pathlib` пропонує більш розвинений та зручний інтерфейс порівняно з `os.path`.

Зазвичай файли на комп'ютері ідентифікують за допомогою ієрархічних шляхів.

Наприклад, можна ідентифікувати файл `wave.txt` на комп'ютері за допомогою цього шляху:
`/Users/sammy/ocean/wave.txt`.

Операційні системи представляють шляхи дещо по-різному.
Windows може представляти шлях до файлу `wave.txt` як
`C:\Users\sammy\ocean\wave.txt`.

Модуль `pathlib` може бути корисним, якщо у програмі Python створюєте або переміщаєте файли у файловій системі, вказуючи всі файли у файловій системі, що збігаються з даним розширенням або шаблоном, або створюєте шляхи файлу, що відповідають файловій системі на основі наборів неформатованих рядків.

Хоча можна використовувати інші інструменти, наприклад модуль `os.path`, для більшої частини цих завдань, але модуль `pathlib` дозволяє виконувати ці операції з більшим ступенем читання та мінімальною кількістю кодів.

Розглянемо деякі способи використання модуля `pathlib` для представлення та маніпуляції шляхами файлових систем.

Модуль `pathlib` надає кілька класів, але одним із найважливіших є клас `Path`.

Примірники класу `Path` представляють шлях до файлу або каталогу файлової системи вашого комп'ютера.

Наприклад, наступний код отримує екземпляр `Path`, який представляє частину шляху до файлу `wave.txt`:

```
from pathlib import Path

wave = Path("ocean", "wave.txt")
print(wave)
```

Якщо запустити цей код, результат буде виглядати так:

```
ocean/wave.txt
```

`from pathlib import Path` робить клас `Path` доступним для нашої програми.

Потім `Path("ocean", "wave.txt")` отримує новий екземпляр `Path`.

З висновку результату видно, що Python «додав» відповідний роздільник оперативної системи між двома заданими нами компонентами шляху "`ocean`" і "`wave.txt`".

Примітка. Залежно від операційної системи, висновок може трохи відрізнятися від прикладів. У Windows, наприклад, висновок цього прикладу може виглядати як `ocean\wave.txt`.

У прикладі об'єкт Path, наданий змінною `wave`, містить відносний шлях.

Можна використовувати `Path.home()` для отримання абсолютного шляху до домашнього каталогу поточного користувача:

```
home = Path.home()
wave_absolute = Path(home, "ocean", "wave.txt")
print(home)
print(wave_absolute)
```

Якщо запустити цей код, результат буде виглядати приблизно так:

```
/Users/sammy
/Users/sammy/ocean/wave.txt
```

Примітка. Як згадувалося раніше, висновок залежатиме від операційної системи

`Path.home()` повертає екземпляр `Path` з абсолютним шляхом до домашнього каталогу поточного користувача.

Потім ми передамо цей екземпляр `Path` і рядки "ocean" і "`wave.txt`" до іншого конструктора `Path`, щоб створити абсолютний шлях до файлу `wave`.txt.

Висновок показує, що перший рядок – це домашній каталог, а другий рядок – домашній каталог плюс `ocean/wave.txt`.

Цей приклад також ілюструє важливу функцію класу `Path`: конструктор `Path` приймає обидві рядки і об'єкти `Path`, що раніше існували.

Детальніше розглянемо підтримку рядків та об'єктів `Path` у конструкторі `Path`:

```
shark = Path(Path.home(), "ocean", "animals", Path("fish",
"shark.txt"))

print(shark)
```

Якщо запустити цей код Python, результат буде виглядати так:

```
/Users/sammy/ocean/animals/fish/shark.txt
```

`shark`— це `Path` до файлу, який ми створили за допомогою об'єктів `Path` (`Path.home()` та `Path("fish", "shark.txt")` та рядків "ocean" та "`animals`").

Конструктор `Path` інтелектуально обробляє обидва типи об'єктів та акуратно з'єднує їх за допомогою відповідного роздільника операційної системи, в даному випадку /.

Доступ до атрибутів файлу

Розглянемо, як можна використовувати екземпляри `Path` для доступу до інформації про файл.

Можемо використовувати атрибути `name` і `suffix` для доступу до імен та розширень файлів:

```
wave = Path("ocean", "wave.txt")
print(wave)
print(wave.name)
print(wave.suffix)
```

Запустивши цей код, отримаємо висновок, аналогічний до наступного:

```
/Users/sammy/ocean/wave.txt
wave.txt
.txt
```

Цей висновок показує, що ім'я файлу в кінці нашого шляху `wave.txt`, а розширення файлу `.txt`.

Примірники `Path` також пропонують функцію `with_name`, що дозволяє без перешкод створювати новий об'єкт `Path` з іншим ім'ям:

```
wave = Path("ocean", "wave.txt")
tides = wave.with_name("tides.txt")
print(wave)
print(tides)
```

Якщо запустити його, результат буде виглядати так:

```
ocean/wave.txt
ocean/tides.txt
```

Код спершу створює екземпляр `Path`, який вказує на файл з ім'ям `wave.txt`.

Потім викликається метод `with_name` у `wave`, щоб повернути другий екземпляр `Path`, що вказує на новий файл з ім'ям `tides.txt`.

Частина каталогу `ocean/` залишається незміненою та залишає фінальний шлях у вигляді `ocean/tides.txt`

Доступ до попередніх об'єктів

Іноді корисно отримати доступ до каталогів, які містять певний шлях. Розглянемо приклад:

```
shark = Path("ocean", "animals", "fish", "shark.txt")
print(shark)
print(shark.parent)
```

Якщо запустити цей код, результат буде виглядати так:

```
ocean/animals/fish/shark.txt
ocean/animals/fish
```

Атрибут parent в екземплярі `Path` повертає найближчого попередника шляху файлу. У цьому випадку він повертає каталог із файлом `shark.txt`: `ocean/animals/fish`.

Можемо отримувати доступ до атрибуту `parent` кілька разів у команді, щоб пройти вгору кореневим деревом даного файлу:

```
shark = Path("ocean", "animals", "fish", "shark.txt")
print(shark)
print(shark.parent.parent)
```

Якщо виконати цей код, побачимо наступні:

```
ocean/animals/fish/shark.txt
ocean/animals
```

Висновок буде схожий на попередній висновок, але тепер ми перейшли на рівень вище, отримавши доступ до .parent вдруге. Два каталоги від `shark.txt` – це каталог `ocean/animals`.

Використання шаблону пошуку для списку файлів

Також можна використовувати клас `Path` для списку файлів за допомогою `glob`.

Припустимо, у нас є структура каталогу, яка виглядає так:

```
└── ocean
    ├── animals
    │   └── fish
    │       └── shark.txt
    ├── tides.txt
    └── wave.txt
```

Каталог ocean містить файли `tides.txt` та `wave.txt`. У нас є файл з ім'ям `shark.txt`, вкладений у каталог ocean, каталог animals та каталог `fish`: `ocean/animals/fish`.

Щоб перерахувати всі файли `.txt` у каталозі `ocean`, можна використовувати:

```
for txt_path in Path("ocean").glob("*.txt"):
  print(txt_path)
```

Цей код зробить такий висновок:

```
ocean/wave.txt
ocean/tides.txt
```

Шаблон пошуку `"*.txt"` знаходить усі файли, що закінчуються на `.txt`. Оскільки приклад коду виконує цей пошук у каталозі ocean, він повертає два файли `.txt` у каталозі ocean: `wave.txt` та `tides.txt`.

Також можна використовувати метод `glob` рекурсивно.

Щоб перерахувати всі файли `.txt` у каталозі ocean і всі його підкаталоги, запишемо:

```
for txt_path in Path("ocean").glob("**/*.txt"):
   print(txt_path)
```

Якщо запустити цей код, результат буде виглядати так:

```
ocean/wave.txt
ocean/tides.txt
ocean/animals/fish/shark.txt
```

Частина ** шаблону пошуку буде відповідати цьому каталогу та всім каталогам під ним рекурсивно.

Тому у висновку у нас будуть не тільки файли `wave.txt` та `tides.txt`, але також ми отримаємо файл `shark.txt`, вкладений у `ocean/animals/fish`.

Обчислення відносних шляхів

Можна використовувати метод `Path.relative_to` для обчислення шляхів, які стосуються один одного. Метод `relative_to` корисний, якщо, наприклад, ви хочете отримати частину довгого шляху файлу.

Розгляньте наступний код:

```
shark = Path("ocean", "animals", "fish", "shark.txt")
below_ocean = shark.relative_to(Path("ocean"))
below_animals = shark.relative_to(Path("ocean",
                                       "animals"))
print(shark)
print(below_ocean)
print(below_animals)
```

Якщо запустити його, результат буде виглядати так:

```
ocean/animals/fish/shark.txt
animals/fish/shark.txt
fish/shark.txt
```

Метод `relative_to` повертає новий об'єкт `Path`, що належить до цього аргументу. У нашому прикладі ми обчислимо `Path` до `shark.txt`, що

відноситься до каталогу ocean, а потім стосується обох каталогів `ocean` і `animals`.

Якщо `relative_to` не зможе вирахувати відповідь, оскільки ми даємо йому не пов'язаний шлях, він видасть `ValueError`:

```
shark = Path("ocean", "animals", "fish", "shark.txt")
shark.relative_to(Path("unrelated", "path"))
```

Ми отримаємо виняток `ValueError`, що виник з цього коду, який виглядатиме так:

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/local/lib/Python3.8/pathlib.py", line 899, in
relative_to
raise ValueError("{!r} does not start with {!r}"
ValueError: 'ocean/animals/fish/shark.txt' does not start with
'unrelated/path'
```

`unrelated/path` не є частиною `ocean/animals/fish/shark.txt`, тому Python не зможе обчислити відносний шлях.

Модуль `pathlib` представляє додаткові класи та утиліти.

Питання для самоконтролю до теми 14

1. Яке призначення модуля pathlib?
2. Як побудувати шлях до файлу за допомогою модуля pathlib?
3. Як дізнатися значення атрибутів файлу за допомогою модуля pathlib?
4. Як дізнатися ім'я попереднього каталогу, використовуючи модуль pathlib?
5. Як встановити шаблон пошуку файлів за допомогою модуля pathlib?

Завдання до теми 14

Напишіть програму пошуку файлів у зазначеній папці та її підпапках за заданим шаблоном. Програма повинна виводити повний шлях до знайдених файлів або повідомляти про відсутність файлів. Ім'я папки та шаблон вказуються як параметри командного рядка виклику програми.

15. МОДУЛЬ `glob`

Модуль `glob` знаходить всі шляхи, що збігаються із заданим шаблоном відповідно до правил, що використовуються оболонкою Unix.

Обробляються символи `"*"` (довільна кількість символів), `"?"` (один символ) та діапазони символів за допомогою `[]`.

Для використання тильди `"~"` та змінних оточення необхідно використовувати `os.path.expanduser()` та `os.path.expandvars()`.

Для пошуку спецсимволів укладайте їх у квадратні дужки. Наприклад, `[?]` відповідає символу "?".

`glob.glob(pathname)`

Повернення списку (можливо, порожній) шляхів, що відповідають шаблону pathname. Шлях може бути абсолютним (наприклад, `/usr/src/Python-1.5/Makefile`) або відносний (`../../Tools/*/*.gif`).

`glob.iglob(pathname)`

Повертає ітератор, що дає ті ж значення, що й `glob.glob`.

`glob.escape(pathname)`

екранує всі спеціальні символи для glob (`"?"`, `"*"` та `"[]"`). Спеціальні символи в імені диска не екрануються (оскільки вони там не враховуються), тобто у Windows `escape("//?/c:/Quo vadis?.txt")` повертає `"//?/c:/Quo vadis[? ].txt"`.

Розглянемо, наприклад, каталог, що містить лише такі файли: `1.gif`, `2.txt` та `card.gif`. glob.glob() поверне наступні результати. (Зверніть увагу, що будь-які провідні компоненти шляху зберігаються):

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
```

Якщо каталог містить файли, що починаються з `"."`, вони не включатимуться за замовчуванням.

Розглянемо, наприклад, каталог, що містить `card.gif` та `.card.gif`:

```
>>>
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

Питання для самоконтролю до теми 15

1. Яке призначення модуля `glob`?
2. Як побудувати шлях до файлу за допомогою модуля `glob`?
3. Як отримати список папок, який відповідає заданому шаблону імені використовуючи модуль glob?
4. Як знайти папки, імена яких містять спеціальні символи за допомогою модуля glob?
5. Як знайти файл у заданій папці та її підпапках використовуючи модуль glob?

Завдання до теми 15

Напишіть програму пошуку в папці під папок, в іменах яких є спеціальні символи. Програма повинна виводити повний шлях до знайдених папок. Ім'я папки вказуються як параметри командного рядка виклику програми.

16. МОДУЛЬ `os.path`

`os.path` є вкладеним модулем модуль `os`, і реалізує деякі корисні функції для роботи з шляхами.

`os.path.abspath(path)` – повертає нормалізований абсолютний шлях.

`os.path.basename(path)` – базове ім'я шляху (еквівалентно `os.path.split(path))`.

`os.path.commonprefix(list)` – повертає найдовший префікс усіх шляхів у списку.

`os.path.dirname(path)` – повертає ім'я директорії шляху `path`.

`os.path.exists(path)` – повертає `True`, якщо `path` вказує на існуючий шлях або дескриптор відкритого файлу.

`os.path.expanduser(path)` – замінює `~` або `~user` на домашню директорію користувача.

`os.path.expandvars(path)` – повертає аргумент із підставленими змінними оточення (`$name` або `${name}` замінюються змінною оточення `name`). Неіснуючі імена не замінює. Windows також замінює `%name%`.

`os.path.getatime(Path)` – час останнього доступу до файлу, в секундах.

`os.path.getmtime(Path)` – час останньої зміни файлу, в секундах.

`os.path.getctime(path)` - час створення файлу (Windows), час останньої зміни файлу (Unix).

`os.path.getsize(Path)` – розмір файлу в байтах.

`os.path.isabs(path)` – чи є шлях абсолютним.

`os.path.isfile(path)` – чи є шлях файлом.

`os.path.isdir(path)` – чи є шлях директорією.

`os.path.islink(path)` – чи є шлях символічним посиланням.

`os.path.ismount(path)` – чи є шлях точкою монтування.

`os.path.join(path1[, path2[, ...]])` – з'єднує шляхи з урахуванням особливостей операційної системи.

`os.path.normcase(path)` – нормалізує регістр шляху (на файлових системах, які не враховують регістр, наводить шлях до нижнього регістру).

`os.path.normpath(path)` – нормалізує шлях, прибираючи надлишкові роздільники та посилання на попередні директорії. На Windows перетворює прямі скісна риска(слеші) на зворотні.

`os.path.realpath(path)` – повертає канонічний шлях, забираючи всі символічні посилання (якщо вони підтримуються).

`os.path.relpath(path, start=None)` – обчислює шлях щодо директорії `start` (за умовчанням – щодо поточної директорії).

`os.path.samefile(path1, path2)` – чи вказують `path1` і `path2` на той самий файл або директорію.

`os.path.sameopenfile(fp1, fp2)` – чи вказують дескриптори `fp1` і `fp2` на той самий відкритий файл.

`os.path.split(path)` – розбиває шлях на кортеж (голова, хвіст), де хвіст – останній компонент шляху, а голова – все інше. Хвіст ніколи не починається зі

скісної риски(слешу) (якщо шлях закінчується скісною рискою(слешем), то хвіст порожній). Якщо скісних рисок(слешів) у шляху немає, то порожньою буде голова.

`os.path.splitdrive(path)` – розбиває шлях на пару (привід, хвіст).

`os.path.splitext(Path)` – розбиває шлях на пару `(root, ext)`, де `ext` починається з точки і містить не більше однієї точки.

`os.path.supports_unicode_filenames` – чи підтримує файлова система Unicode.

Питання для самоконтролю до теми 16

1. Яке призначення модуля `os.path`?
2. Як отримати абсолютний шлях до файлу, використовуючи модуль `os.path`?
3. Як дізнатися останнім часом доступу до файлу, використовуючи модуль `os.path`?
4. Як дізнатися розмір файлу, використовуючи модуль `os.path`?
5. Як дізнатися, чи використовується в ОС кодування Unicode використовуючи модуль `os.path`?

Завдання до теми 16

Напишіть програму пошуку файлів у зазначеній папці та її підпапках розмір яких лежить у заданому інтервалі. Програма повинна виводити повний шлях до знайдених файлів та їх розмір або повідомляти про відсутність файлів. Ім'я папки та мінімальний та максимальний розмір файлу вказуються як параметри командного рядка виклику програми.

17. Модуль sys

Модуль sys забезпечує доступ до деяких змінних та функцій, що взаємодіють з інтерпретатором python.

sys.argv – перелік аргументів командного рядка, що передаються сценарієм Python. sys.argv[0] є ім'ям скрипту (порожнім рядком в інтерактивній оболонці).

Приклад: вивести усі аргументи командного рядка:

```
for param in sys.argv:
   print(param)
```

sys.byteorder – порядок байтів. Матиме значення 'big' при порядку проходження бітів від старшого до молодшого, і 'little', якщо навпаки (молодший байт перший).

sys.builtin_module_names – кортеж рядків, що містить імена всіх доступних модулів.

sys.call_tracing(функція, аргументи) – викликає функцію з аргументами та включеним трасуванням, у той час як трасування включене.

sys.copyright – рядок, що містить авторські права, що відносяться до інтерпретатора Python.

sys._clear_type_cache() – очищає внутрішній кеш типів.

sys._current_frames() – повертає словник відображення ідентифікатора для кожного потоку у верхньому кадрі стека в даний час у цьому потоці в момент виклику функції.

sys.dllhandle – ціле число, що визначає дескриптор DLL Python (Windows).

sys.exc_info() – повертає кортеж із трьох значень, які дають інформацію про винятки, що опрацьовуються на даний момент.

sys.exec_prefix – каталог встановлення Python.

sys.executable – шлях до інтерпретатора Python.

sys.exit([arg]) – вихід із Python. Порушує виняток SystemExit, який може бути перехоплений.

sys.flags – прапорці командного рядка. Атрибути лише для читання.

sys.float_info – інформація про тип даних float.

sys.float_repr_style – інформація про застосування вбудованої функції repr() типу float.

sys.getdefaultencoding() – повертає використовуване кодування.

sys.getdlopenflags() – значення прапорів для викликів dlopen().

sys.getfilesystemencoding() – повертає кодування файлової системи.

sys.getrefcount(object) – повертає кількість посилань на об'єкт. Аргумент функції getrefcount – ще одне посилання на об'єкт.

sys.getrecursionlimit() – повертає ліміт рекурсії.

sys.getsizeof(object [, default]) – повертає розмір об'єкта (в байтах).

sys.getswitchinterval() – інтервал перемикання потоків.

sys.getwindowsversion() – повертає кортеж, що описує версію Windows.

sys.hash_info – інформація про параметри хешування.

`sys.hexversion` – версія python як шістнадцяткове число (для 3.2.2 final це буде 30202f0).

`sys.implementation` – об'єкт, що містить інформацію про запущеного python інтерпретатора.

`sys.int_info` – інформація про тип `int`.

`sys.intern(рядок)` – повертає інтернований рядок.

`sys.last_type, sys.last_value, sys.last_traceback` – інформація про оброблені винятки. За змістом схоже `sys.exc_info()`.

`sys.maxsize` – максимальне значення числа типу `Py_ssize_t` (231 на 32-бітових та 263 на 64-бітних платформах).

`sys.maxunicode` – максимальна кількість біт для зберігання символу Unicode.

`sys.modules` – словник імен завантажених модулів. Змінюємо, тому можна потішитися.

`sys.path` – перелік шляхів пошуку модулів.

`sys.path_importer_cache` – словник-кеш для пошуку об'єктів.

`sys.platform`– інформація про операційну систему.

Linux (2.x і 3.x)	'linux'
Windows	'win32'
Windows/Cygwin	'cygwin'
Mac OS X	'darwin'
OS/2	'os2'
OS/2 EMX	'os2emx'

`sys.prefix` – папка встановлення інтерпретатора python.

`sys.ps1, sys.ps2` – первинне та вторинне запрошення інтерпретатора (визначено лише якщо інтерпретатор перебуває в інтерактивному режимі). За замовчанням `sys.ps1 == ">>> "`, а `sys.ps2 == "... "`.

`sys.dont_write_bytecode` – якщо `true`, python не писатиме `.pyc` файли.

`sys.setdlopenflags(flags)` – встановити значення прапорів для викликів `dlopen()`.

`sys.setrecursionlimit(між)` – встановити максимальну глибину рекурсії.

`sys.setswitchinterval(інтервал)` – встановити інтервал перемикання потоків.

`sys.settrace(tracefunc)` – встановити "слід" функції.

`sys.stdin` – Стандартне введення.

`sys.stdout` – стандартний висновок.

`sys.stderr` – стандартний потік помилок.

`sys.__stdin__, sys.__stdout__, sys.__stderr__` – вихідні значення потоків введення, виведення та помилок.

`sys.tracebacklimit` – максимальна кількість рівнів відстеження.

`sys.version` – версія python.

`sys.api_version` – версія с API.

`sys.version_info` – кортеж, який містить п'ять компонентів номера версії.

`sys.warnoptions` – реалізація попереджень.

`sys.winver` – Номер версії python, який використовується для формування реєстру Windows.

Питання для самоконтролю до теми 17

1. Яке призначення модуля `sys`?
2. Як отримати список аргументів командного рядка, які передаються сценарієм Python?
3. Як знайти шлях до інтерпретатора Python?
4. Як дізнатися використовуване кодування?
5. Як отримати інформацію про операційну систему, що використовується?

Завдання до теми 17

Напишіть програму яка створює список та кортеж, використовуючи "генератор списків" (`list comprehensions`) та "генератор кортежів" (`generator expressions`). Наприклад:

```
my_list = [i * 2 for i in range(1000)]
```

- створює список із 1000 елементів відразу і

```
my_generator = (i * 2 for i in range(1000))
```

- створює генератор, який почне робити значення лише за ітерації.

Знайдіть розмір об'єктів `list_comp` та `my_generator` в байтах. Порівняйте ці розміри. Поясніть результати порівняння.

Модуль `itertools`

Модуль `itertools` – збірка корисних ітераторів.

`itertools.count(start = 0, step = 1)` – нескінченна арифметична прогресія з першим членом start та кроком step.

`itertools.cycle(iterable)` – повертає по одному значенню з послідовності, повтореної нескінченне число разів.

`itertools.repeat(Elem, n = Inf)` – повторює elem n разів.

`itertools.accumulate(iterable)` – акумулює суми: `accumulate([1,2,3,4,5])` --> 1 3 6 10 15

`itertools.chain(*iterables)` – повертає по одному елементу з першого ітератора, потім з другого, доки ітератори не закінчаться.

`itertools.combinations(iterable, [r])` – комбінації довжиною `r` з `iterable` без повторюваних елементів.

`combinations('ABCD', 2)` --> AB AC AD BC BD CD

`itertools.combinations_with_replacement(iterable, r)` – комбінації довжиною `r` з `iterable` з елементами, що повторюються.

`combinations_with_replacement('ABCD', 2)` -->

AA AB AC AD BB BC BD CC CD DD

`itertools.compress(data, selectors)` –

`(d[0] if s[0]), (d[1] if s[1]), ...`

compress('ABCDEF', [1,0,1,0,1,1]) --> ACEF

`itertools.dropwhile(func, iterable)` – елементи `iterable`, починаючи з першого, для якого `func` повернула `False`.

`dropwhile(lambda x: x < 5, [1,4,6,4,1]) --> 6 4 1`

`itertools.filterfalse(func, iterable)` – всі елементи, для яких func повертає `False`.

`itertools.groupby(iterable, key=None)` – групує елементи за значенням. Значення виходить застосуванням функції `key` до елемента (якщо аргумент `key` не вказано, значенням є сам елемент).

```
>>>
>>> from itertools import groupby
>>> things = [("animal", "bear"), ("animal", "duck"),
                                        ("plant", "cactus"),
         ... ("vehicle", "speed boat"), ("vehicle", "school bus")]

>>> for key, group in groupby(things, lambda x: x[0]):
..  . for thing in group:
..      . print("A %s is a %s." % (thing[1], key))
...   print()
Bear is a animal.
Duck is a animal.

A cactus is a plant.

A speed boat is a vehicle.
A school bus is a vehicle.
```

`itertools.islice(iterable[, start], stop[, step])` – ітератор, що складається із зрізу.

`itertools.permutations(iterable, r=None)` – перестановки завдовжки `r` з `iterable`.

Приклад.

```
a=[1,2,3]
b=list(itertools.permutations(a))
print(b)
[(1, 2, 3), (1, 3, 2), (2, 1, 3),
(2, 3, 1), (3, 1, 2), (3, 2, 1)]
```

`itertools.product(*iterables, repeat=1)` – аналог вкладених циклів.

`product('ABCD', 'xy')` --> Ax Ay Bx By Cx Cy Dx Dy

`itertools.starmap(function, iterable)` – застосовує функцію кожного елемента послідовності (кожен елемент розпаковується).

`starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000`

`itertools.takewhile(func, iterable)` – елементи до тих пір, поки func повертає істину.

`takewhile(lambda x: x<5, [1,4,6,4,1])` --> 1 4

`itertools.tee(iterable, n=2)` – кортеж з `n` ітераторів.

`itertools.zip_longest(*iterables, fillvalue=None)` – як вбудована функція `zip`, але бере найдовший ітератор, а короткі доповнює `fillvalue`.

`zip_longest('ABCD', 'xy', fillvalue='-')` --> Ax By C-D-

Питання для самоконтролю до теми 18

1. Що таке ітератор у програмі Python?
2. Як отримати всі комбінації заданої довжини з ітерованого списку елементів без повторюваних елементів
3. Як застосувати функцію до кожного елемента послідовності
4. Як знайти всі перестановки довжиною r із заданого списку, використовуючи модуль `itertools`?

Завдання до теми 18

Напишіть скрипт, який знаходить всі перестановки елементів заданого списку у вигляді списку. Виведіть знайдену кількість перестановок. Перетворіть на список кортежів. Виконайте сортування елементів цього списку та організуйте виведення елементів списку на консоль.

19. МОДУЛЬ `locale`

При форматуванні чисел Python за замовчуванням використовує англосаксонську систему, при якій розряди цілого числа відокремлюються один від одного комами, а частина від цілої відокремлюється точкою.

У континентальній Європі, наприклад, використовується інша система, за якої розряди розділяються точкою, а дробова і ціла частина – комою:

англосаксонська система

```
1,234.567
```

європейська система

```
1.234,567
```

Для вирішення проблеми форматування під певну локаль Python є вбудований модуль `locale`.

Для з'ясування яка локаль встановлена можна виконати:

```
import locale
locale.getlocale()
```

Для встановлення локалі в модулі `locale` визначено функцію `setlocale()`. Вона приймає два параметри:

```
setlocale(category, locale)
```

Перший параметр вказує на категорію, до якої застосовується функція – до чисел, валют або числах, і валют. Як значення для параметра можна передавати одну з наступних констант:

- `locale.LC_ALL`: застосовує локалізацію до всіх категорій – до форматування чисел, валют, дат тощо.
- `locale.LC_NUMERIC`: застосовує локалізацію до чисел
- `locale.LC_MONETARY`: застосовує локалізацію до валют
- `locale.LC_TIME`: застосовує локалізацію до дат та часу
- `locale.LC_CTYPE`: застосовує локалізацію під час перекладу символів у верхній або нижній регістр
- `locale.LC_COLLIATE`: застосовує локаль для порівняння рядків

Другий параметр функції `setlocale` вказує на локаль, яку потрібно використовувати.

На Windows можна використовувати код країни за ISO із двох символів:
для США – "us",
для Німеччини – "de",

Але на MacOS необхідно вказувати код мови та код країни, наприклад, для англійської в США – `"en_US"`, для німецької в Німеччині – `"de_DE"`. За замовчанням фактично використовується локаль `"en_US"`.

Безпосередньо для форматування чисел та валют модуль `locale` надає дві функції:

- `currency(num)` - форматує валюту
- `format(str, num)` – підставляє число num замість заповнювача в рядок `str`

Застосовуються такі заповнювачі:

- `d`: для цілих чисел
- `f`: для чисел з рухомою комою
- `e`: для експоненційного запису чисел

Перед кожним заповнювачем ставиться знак відсотка `%`, наприклад:`"%d"`

При виведенні дробових чисел перед заповнювачем після точки можна вказати скільки знаків у дробовій частині повинно відображатися:

```
%.2f # два знаки в дрібній частині
```

Застосуємо локалізацію чисел та валют у німецькій локалі:

```
import locale

locale.setlocale(locale.LC_ALL, "de") # для Windows
# locale.setlocale(locale.LC_ALL, "de_DE")# для MacOS

number = 12345.6789
formatted = locale.format("%f", number)
print(formatted) # 12345,678900

formatted = locale.format("%.2f", number)
print(formatted) # 12345,68

formatted = locale.format("%d", number)
print(formatted) # 12345

formatted = locale.format("%e", number)
print(formatted) # 1,234568e+04

гроші = 234.678
formatted = locale.currency(money)
print(formatted) # 234,68 €
```

Якщо замість конкретного коду як другий параметр функції `setlocale()` передати порожній рядок, то Python буде використовувати локаль, що застосовується на поточній робочій машині.

За допомогою функції `getlocale()` можна дізнатися про поточну локаль:

```
import locale
```

```
    locale.setlocale(locale.LC_ALL, "")
    number = 12345.6789
    formatted = locale.format_string("%.02f", number)
    print(formatted)
    print(locale.getlocale())
```

Приклад зміни локалі та отримання списків імен локалей:

```
import locale
locale.setlocale(locale.LC_ALL, "uk_ua")
number = 12345.6789
formatted = locale.format_string("%.02f", number)
print(formatted)
print(locale.getlocale())
print(locale.localeconv())
print(50*'*')
#print(locale.locale_encoding_alias)
print(50*'*')
for key, value in ocale.locale_encoding_alias.items():
        print(f"name: {key}  alias {value}")
print(50*'*')
for key, value in locale.locale_alias.items():
        print(f"key: {key}  value {value}")
```

Питання для самоконтролю до теми 19

1. Що таке локаль операційної системи?
2. Яке призначення модуля locale?
3. Як дізнатися, яка локаль встановлена в ОС?
4. Як змінити локаль для скрипту Python, що виконується?
5. Які коди країн ви знаєте?

Завдання до теми 19

Напишіть скрипт, який отримує список імен локалей. Організуйте виведення елементів списку на консоль.

20. МОДУЛЬ `datetime`

Основний функціонал для роботи з датами та часом зосереджений у модулі datetime у вигляді наступних класів.

Клас `date`

Для роботи з датами скористаємося класом `date`, визначеним у модулі `datetime`. Для створення об'єкта `date` ми можемо використовувати конструктор `date`, який послідовно приймає три параметри: рік, місяць та день:

```
date(year, month, day)
```

Наприклад, створимо якусь дату:

```
import datetime

yesterday = datetime.date(2017,5, 2)
print(yesterday) # 2017-05-02
```

Якщо необхідно отримати поточну дату, можна скористатися методом `today()`:

```
from datetime import date

today = date.today()
print(today) # 2017-05-03
print("{}.{}.{}".format(today.day, today.month,
                                today.year)) # 2.5.2017
```

За допомогою властивостей `day, month, year` можна отримати відповідно день, місяць та рік

Клас `time`

За роботу з часом відповідає клас часу. Використовуючи його конструктор, можна створити об'єкт часу:

```
time([hour] [, min] [, sec] [, microsec])
```

Конструктор послідовно приймає години, хвилини, секунди та мікросекунди. Усі параметри необов'язкові, і якщо ми якийсь параметр не передамо, то відповідне значення ініціалізуватиметься нулем.

```
from datetime import time
current_time = time()
print(current_time)    # 00:00:00
current_time = time(16, 25)
print(current_time)    # 16:25:00
current_time = time(16, 25, 45)
print(current_time)    # 16:25:45
```

Клас `datetime`

Клас `datetime` з однойменного модуля поєднує можливості роботи з датою та часом. Для створення об'єкта datetime можна використовувати наступний конструктор:

```
datetime(year, month, day [, hour] [, min] [, sec] [, microsec])
```

Перші три параметри, що становлять рік, місяць і день, є обов'язковими. Інші необов'язкові, і якщо ми не вкажемо для них значення, то за умовчанням вони ініціалізуються банкрутом.

```
from datetime import datetime
deadline = datetime(2017, 5, 10)
print(deadline)     # 2017-05-10 00:00:00

deadline = datetime(2017, 5, 10, 4, 30)
print(deadline) # 2017-05-10 04:30:00
```

Для отримання поточної дати та часу можна викликати метод now():

```
from datetime import datetime

now = datetime.now()
print(now)              # 2017-05-03 11:18:56.239443

print("{}.{}.{} {}:{}".format(now.day, now.month,
       now.year, now.hour, now.minute)) # 3.5.2017 11:21

print(now.date())
print(now.time())
```

За допомогою властивостей `day, month, year, hour, minute, second` можна отримати окремі значення дати та часу. А через методи `date()` та `time()` можна отримати окремо дату та час відповідно.

Приклад:

```
from datetime import datetime
import locale
locale.setlocale(locale.LC_ALL, "uk-ua")
now = datetime.now()
print("локаль: uk_ua", now.strftime("%d %B %Y (%A)"))
locale.setlocale(locale.LC_ALL, "de")
now1 = datetime.now()
print("локаль: de", now1.strftime("%d %B %Y (%A)"))
```

Результат виконання скрипту:

```
локаль: uk_ua 30 жовтень 2025 (четвер)
локаль: de 30 Oktober 2025 (Donnerstag)
```

Перетворення з рядка на дату

З функціональності класу `datetime` слід зазначити метод `strptime()`, який дозволяє розпарсити рядок та перетворити його на дату. Цей метод приймає два параметри: `strptime(str, format)`

Перший параметр `str` представляє рядкове визначення дати та часу, а другий параметр – формат, який визначає, як різні частини дати та часу розташовані у цьому рядку.

Для визначення формату можна використовувати такі коди:

- `%d`: день місяця у вигляді числа
- `%m`: порядковий номер місяця
- `%y`: рік у вигляді 2-х чисел
- `%Y`: рік у вигляді 4-х чисел
- `%H`: година у 24-х годинному форматі.
- `%M`: хвилина
- `%S`: секунда

Застосуємо різні формати:

```
from datetime import datetime
deadline = datetime.strptime("22/05/2017", "%d/%m/%Y")
print(deadline) # 2017-05-22 00:00:00

deadline = datetime.strptime("22/05/2017 12:30", "%d/%m/%Y %H:%M")
print(deadline) # 2017-05-22 12:30:00

deadline = datetime.strptime("05-22-2017 12:30","%m-%d-%Y %H:%M")
print(deadline) # 2017-05-22 12:30:00
```

Операції з датами

Форматування дат та часу.

Для форматування об'єктів `date` і `time` в обох класах передбачений метод `strftime(format)`.

Цей метод приймає лише один параметр, що вказує на формат, який потрібно перетворити дату або час.

Для визначення формату можна використовувати один із таких кодів форматування:

- `%a`: абревіатура дня тижня. Наприклад, Wed – від слова Wednesday (за замовчуванням використовуються англійські найменування)
- `%A`: день тижня повністю, наприклад, Wednesday
- `%b`: абревіатура назви місяця. Наприклад, Oct (скорочення від October)
- `%B`: назва місяця повністю, наприклад, October
- `%d`: день місяця, доповнений нулем, наприклад, 01
- `%m`: номер місяця, доповнений нулем, наприклад, 05

- `%y`: рік у вигляді 2-х чисел
- `%Y`: рік у вигляді 4-х чисел
- `%H`: година у 24-х годинному форматі, наприклад, 13
- `%I`: година у 12-ти годинному форматі, наприклад, 01
- `%M`: хвилина
- `%S`: секунда
- `%f`: мікросекунда
- `%p`: покажчик AM/PM
- `%c`: дата та час, відформатовані під поточну локаль
- `%x`: дата, відформатована під поточну локаль
- `%X`: час, форматований під поточну локаль

Використовуємо різні формати:

```
from datetime import datetime
now = datetime.now()
print(now.strftime("%Y-%m-%d")) # 2017-05-03
print(now.strftime("%d/%m/%Y")) # 03/05/2017
print(now.strftime("%d/%m/%y")) # 03/05/17
print(now.strftime("%d %B %Y (%A)")) # 03 May 2017
                                     # (Wednesday)

print(now.strftime("%d/%m/%y %I:%M")) 03/05/17 01:36
```

При виведенні назв місяців та днів тижня за умовчанням використовуються англійські найменування. Якщо ми хочемо використовувати поточну локаль, але ми можемо її попередньо встановити за допомогою модуля `locale`:

```
from datetime import datetime
import locale
locale.setlocale(locale.LC_ALL, "uk_ua")

now = datetime.now()
print(now.strftime("%d %B %Y (%A)"))
```

Результат виконання скрипту:

```
30 жовтень 2025 (четвер)
```

Складання та віднімання дат і часу

Нерідко при роботі з датами виникає необхідність додати до будь-якої дати певний проміжок часу або, навпаки, відняти певний період. І спеціально для таких операцій в модулі `datetime` визначено клас `timedelta`. Фактично цей клас визначає певний період.

Для визначення проміжку часу можна використовувати конструктор `timedelta`:

```
timedelta([days] [, seconds] [, microseconds]
                [, milliseconds] [, minutes] [, hours] [, weeks])
```

У конструктор послідовно передаються дні, секунди, мікросекунди, мілісекунди, хвилини, години та тижні.

Визначимо кілька періодів:

```
from datetime import timedelta
three_hours = timedelta(hours=3)
print(three_hours) # 3:00:00
three_hours_thirty_minutes=timedelta(hours=3, minutes=30) #3:30:00

two_days = timedelta(2)      # 2 days, 0:00:00

two_days_three_hours_thirty_minutes = timedelta(days=2,
                          hours=3, minutes=30) # 2 days, 3:30:00
```

Використовуючи `timedelta`, можна складати або віднімати дати. Наприклад, отримаємо дату, яка буде за два дні:

```
from datetime import timedelta, datetime
now = datetime.now()
print(now)
two_days = timedelta(2)
in_two_days = now + two_days
print(in_two_days)
```

Результат виконання скрипту:

```
2018-03-06 12:20:48.524307
2018-03-08 12:20:48.524307
```

Або дізнаємося, скільки було часу 10 годин 15 хвилин тому, тобто фактично нам треба відняти з поточного часу 10 годин і 15 хвилин:

```
from datetime import timedelta, datetime
now = datetime.now()
till_ten_hours_fifteen_minutes = now -
                                  timedelta(hours=10, minutes=15)
print(till_ten_hours_fifteen_minutes)
```

Властивості `timedelta`

Клас `timedelta` має кілька властивостей, за допомогою яких ми можемо отримати часовий проміжок:

- `days`: повертає кількість днів
- `seconds`: повертає кількість секунд
- `microseconds`: повертає кількість мікросекунд.

Крім того, метод `total_seconds()` повертає загальну кількість секунд, куди входять і дні, і секунди, і мікросекунди.

Наприклад, дізнаємося якийсь тимчасовий період між двома датами:

```
from datetime import timedelta, datetime

now = datetime.now()
```

```
twenty_two_may = datetime(2017, 5, 22)
period = twenty_two_may - now
print("{} днів {} секунд {} мікросекунд". format(period.days,
period.seconds, period.microseconds))
# 18 днів 17537 секунд 72765 мікросекунд

print("Всього: {} секунд". format(period.total_seconds()))

# Всього: 1572737.072765 секунд
```

Порівняння дат

Так само як і рядки та числа, дати можна порівнювати за допомогою стандартних операторів порівняння:

```
from datetime import datetime
now = datetime.now()
deadline = datetime(2017, 5, 22)
if now > deadline:
print("Термін здачі проєкту пройшов")
elif now.day == deadline.day and now.month == deadline.month and
now.year == deadline.year:
print("Термін здачі проєкту сьогодні")
else:
period = deadline - now
print("Залишилось {} днів".format(period.days))
```

Питання для самоконтролю до теми 20.

1. Яке призначення модуля `datetime`?
2. Як створити об'єкт класу `date`?
3. Як створити об'єкт класу часу?
4. Як перетворення символьного рядка в дату?
5. Які операції з датами реалізовані в модулі `datetime`?

Завдання до теми 20

Напишіть скрипт, який запитує дату народження користувача та виводить кількість прожитих днів.

21. МОДУЛЬ `logging`

Уявіть ситуацію, коли необхідно зберегти деякі налагоджувальні або інші важливі повідомлення де-небудь, щоб мати можливість пізніше перевірити, чи програма відпрацювала, як очікувалося. Як “зберегти десь” ці повідомлення?

Зробити це можна за допомогою модуля `logging`.

```
import os,platform, logging

if platform.platform().startswith('Windows'):
  logging_file = os.path.join(os.getenv('HOMEDRIVE'), \
     os.getenv('HOMEPATH'), \
       'test.log')
else:
  logging_file = os.path.join(os.getenv('HOME'),
                                          'test.log')

print("Зберігаємо журнал подій в",logging_file)

logging.basicConfig( \
      level=logging.DEBUG,  \
      format='%(asctime)s : %(levelname)s : %(message)s',  \
      filename = logging_file, filemode = 'w',)

logging.debug("Початок програми")
logging.info("Якісь дії")
logging.warning("Програма вмирає")
```

Результат виконання:

```
$python3 use_logging.py
Зберігаємо журнал подій в C:\Users\bsu\test.log
```

Якщо відкрити файл `test.log`, він виглядатиме приблизно так:

```
2018-03-06 12:31:35,033 : DEBUG : Початок програми
2018-03-06 12:31:35,033 : INFO : Якісь дії
2018-03-06 12:31:35,033 : WARNING : Програма вмирає
```

Як це працює:

Використовувалися три модулі зі стандартної бібліотеки: модуль `os` для взаємодії з операційною системою, модуль platform для отримання інформації про платформу (тобто операційну систему) та модуль logging для збереження журналу подій.

Насамперед, за допомогою рядка, що повертається функцією `platform.platform()`, перевіряємо, яка операційна система використовується (для більш детальної інформації див. `import platform; help(platform)`). Якщо це Windows, то ми визначаємо диск, що містить домашній каталог, шлях до домашнього каталогу на ньому та ім'я файлу, в якому хочемо зберегти інформацію. Склавши всі ці три частини, отримуємо повний шлях до файлу. Для інших платформ нам потрібно знати шлях до домашнього каталогу користувача, і отримаємо повний шлях до файлу.

За допомогою функції `os.path.join()` поєднуємо три частини шляху до файлу разом. Використовуємо цю функцію замість простого об'єднання рядків, щоб гарантувати, що повний шлях до файлу записаний у форматі, що очікується операційною системою.

Далі ми конфігуруємо модуль `logging` таким чином, щоб він записував усі повідомлення у певному форматі у вказаний файл.

Нарешті, можемо виводити повідомлення, призначені для налагодження, інформування, попередження та навіть критичні повідомлення. Після виконання програми можна переглянути цей файл і дізнатися, що відбувалося в програмі, хоча користувачу, який запустив програму, нічого не було показано.

Питання для самоконтролю до темі 21

1. Яке призначення модуля `logging`?
2. Як вивести інформаційне повідомлення, використовуючи модуль `logging`?
3. Які формати повідомлень реалізують модуль `logging`?
4. Як задати файл для запису повідомлень до журналу події?
5. Як здійснити базову конфігурацію модуля `logging`?

Завдання до теми 21

Напишіть клас, метод якого виводить повідомлення, що записуються до журналу подій, в заданий файл і на консоль. Перший параметр функції – символ, що визначає вид повідомлення та дію функції. Передбачити такі допустимі символи:

`d` введення повідомлення типу `debug`;

`i` введення повідомлення типу `info`;

`w` введення повідомлення типу `warning`;

`a` введення повідомлення про аварійне завершення програми. Програма має бути завершена.

Другий параметр – текст повідомлення. Повідомлення виводитиме у форматі модуля `logging`. Конструктор класу відкриває файл журналу подій.

22. СТВОРЕННЯ GUI ЗА ДОПОМОГОЮ БІБЛІОТЕКИ `tkinter`

Введення в `tkinter`

У різноманітті програм, які пишуть програмісти, виділяють додатки з графічним інтерфейсом користувача (GUI).

При створенні таких програм стають важливими не лише алгоритми обробки даних, але й розробка для користувача програми зручного інтерфейсу, взаємодіючи з яким він визначатиме поведінку програми.

Сучасний користувач в основному взаємодіє з програмою за допомогою різних кнопок, меню, значків, вводячи інформацію в спеціальні поля, вибираючи певні значення у списках та ін. (від англ. Widget – "штучка").

Для мови програмування Python такі віджети включені до спеціальної бібліотеки `tkinter`. Якщо її імпортувати в програму (скрипт), можна користуватися її компонентами, створюючи графічний інтерфейс.

Послідовність кроків під час створення графічного застосування має свої особливості. Програма має виконувати своє основне призначення, бути зручною для користувача, реагувати на його дії. Ми не будемо вдаватися до подробиць розробки, а розглянемо які етапи приблизно потрібно пройти при програмуванні, щоб отримати програму з GUI:

1. Імпорт бібліотеки
2. Створення головного вікна
3. Створення віджет
4. Встановлення їх властивостей
5. Визначення подій
6. Визначення обробників подій
7. Розташування віджет на головному вікні
8. Відображення головного вікна

Імпорт модуля `tkinter`

Як і будь-який модуль, `tkinter` у Python можна імпортувати двома способами: командами `import tkinter` або `from tkinter import *`.

Надалі користуватимемося лише другим способом, тому що це дозволить не вказувати щоразу ім'я модуля при зверненні до об'єктів, які в ньому містяться. Слід звернути увагу, що у версії Python 3 ім'я модуля пишеться з малої літери (tkinter), хоча в попередніх версіях використовувалася велика (Tkinter). Отже, перший рядок програми має виглядати так:

```
from tkinter import *
```

Створення головного вікна

У сучасних операційних системах будь-який додаток користувача укладено у вікно, яке можна назвати головним, тому що у ньому розташовуються решта

віджетів. Об'єкт верхнього рівня вікна створюється при зверненні до класу Tk модуля tkinter. Змінну пов'язану з об'єктом-вікном прийнято називати root (хоча зрозуміло, що можна назвати як завгодно, але вже прийнято). Другий рядок коду:

```
root = Tk()
```

Створення віджет

Допустимо у вікні розташовуватиметься лише одна кнопка. Кнопка створюється при зверненні до класу `Button tkinter` модуля. Об'єкт кнопка зв'язується з якоюсь змінною. У класу `Button` (як і решти класів, крім `Tk`) є обов'язковий параметр — об'єкт, якому кнопка належить (кнопка неспроможна «бути нічийної»). Поки маємо єдине вікно (`root`), воно і буде аргументом, що передається в клас при створенні об'єкта-кнопки:

```
but = Button(root)
```

Встановлення властивостей віджет

Кнопка має багато властивостей: розмір, колір фону та написи та ін. Для прикладу встановимо лише одну властивість — текст напису (`text`):

```
but["text"] = "Друк"
```

Визначення подій та їх обробників

Різноманітність подій та способів їхньої обробки буде розглянуто далі. Тут же просто торкнемося цього питання у зв'язку з потребою.

Що ж робитиме кнопка і в який момент вона це робитиме? Припустимо, що завдання кнопки вивести якесь повідомлення у потік виведення, використовуючи функцію `print`. Робити вона це при натисканні на неї лівою кнопкою миші.

Дії (алгоритм), які відбуваються при тій чи іншій події, можуть бути досить складними. Тому часто їх оформляють як функції, а потім викликають, коли вони знадобляться.

Нехай у нас друк на екран буде оформлений у вигляді функції `printer`:

```
def printer(event):
  print ("Як завжди черговий 'Hello World!'")
```

Не забувайте, що функцію бажано (майже обов'язково) розміщувати на початку коду. Параметр `event` – це подія.

Подія натискання лівою кнопкою миші виглядає так: `<Button-1>`. Потрібно пов'язати цю подію з обробником (функцією `printer`). Для зв'язку призначено метод `bind`.

Синтаксис зв'язування події з обробником виглядає так:

```
but.bind("<Button-1>",printer)
```

Подія – змінився розмір вікна:

```
# обробник res закриття вікна
root.bind('<Configure>',res)
```

В обробнику можна отримати новий розмір графічного вікна:

```
str=root.geometry()
k1=str.index('x')
xmax1=str[:k1]
k2=str.index('+',k1+1)
ymax1=str[k1+1:k2]
xmax = int (xmax1)
ymax=int(ymax1)
```

Подія – закрити вікно:

```
# обробник window_deleted закриття вікна
root.protocol('WM_DELETE_WINDOW', window_deleted)
```

Розміщення віджет

В будь-якому додатку віджети не розкидані по вікну аби як, а добре організовані. Необхідно вміти кнопку та і інші віджети відобразити у вікні там де хоче програміст. Найпростіший спосіб – це використання методу `pack -`

```
but.pack()
```

Якщо не вставити цей рядок коду, то кнопка у вікні так і не з'явиться, хоча вона є в програмі.

Відображення головного вікна

Ну і нарешті, головне вікно теж не з'явиться, доки не буде викликаний спеціальний метод mainloop:

```
root.mainloop()
```

Цей рядок коду повинен бути завжди в кінці скрипту!

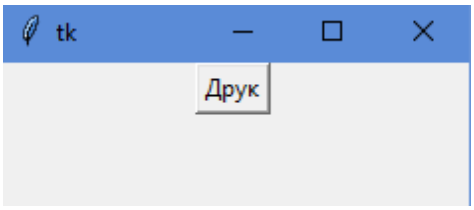
У результаті код програми може виглядати таким чином:

```
from tkinter import *

def printer(event):
   print ("Як завжди черговий 'Hello World!'")

root = Tk()
but = Button(root)
but["text"] = "Друк"
but.bind("<Button-1>",printer)

but.pack()
root.mainloop()
```

При програмуванні графічного інтерфейсу користувача ефективнішим виявляється об'єктно-орієнтований підхід. Тому багато «речей» оформляються у вигляді класів. У нашому прикладі можна також використовувати клас:

```
from tkinter import *

class But_print:
   def __init__(self):
      self.but = Button(root)
      self.but["text"] = "Друк"
```

```
        self.but.bind("<Button-1>",self.printer)
        self.but.pack()
    def printer(self,event):
        print ("Як завжди черговий 'Hello World!'")

root = Tk()
obj = But_print()
root.mainloop()
```



Розмітка віджетів у Tkinter — pack, grid та place

Познайомимось із менеджерами розмітки. Коли створюємо графічний інтерфейс нашої програми, визначаємо, які віджети будемо використовувати і як вони будуть розташовані в додатку. Щоб організувати віджети у додатку, використовуються спеціальні невидимі об'єкти – менеджери розмітки.

Існує два види віджетів:

- контейнери;
- дочірні віджети.

Контейнери поєднують віджети для формування розмітки. У `Tkinter` є три вбудовані менеджери розмітки: `pack`, `grid` і `place`.

- `place`– це менеджер геометрії, який розміщує віджети, використовуючи абсолютну позиціонування.
- `pack`– це менеджер геометрії, який розміщує віджети по горизонталі та вертикалі.
- `grid`– це менеджер геометрії, який розміщує віджети у двовимірній сітці.

Метод `place()` абсолютне позиціонування

Найчастіше розробникам потрібно використовувати менеджери розмітки. Є кілька ситуацій, у яких слід використати саме абсолютне позиціонування. В рамках абсолютного позиціонування розробник визначає позицію та розмір кожного віджету в пікселях. Під час зміни розмірів вікна розмір та позиція віджетів не змінюються.

Зображення для прикладу:bardejov.jpg rotunda.jpg mincol.jpg. Збережіть у папці поруч із файлом absolute.py код для якого буде нижче.

Для цього скрипту необхідно встановити пакет Image:

```
python -m pip install Image
```

Таким чином, на різних платформах програми виглядають по-різному. Те, що виглядає нормально на Linux, може некоректно відображатися на Mac OS. Зміна шрифтів у додатку може також зіпсувати розмітку.

```
from PIL import Image, ImageTk
from tkinter import Tk, BOTH
from tkinter.ttk import Frame, Label, Style

class Example(Frame):

    def __init__(self):
        super().__init__()

        self.initUI()


    def initUI(self):

        self.master.title("Absolute positioning")
        self.pack(fill=BOTH, expand=1)

        Style().configure("TFrame", background="#333")

        bard = Image.open("bardejov.jpg")
        bardejov = ImageTk.PhotoImage(bard)
        label1 = Label(self, image=bardejov)
        label1.image = bardejov
        label1.place(x=20, y=20)

        rot = Image.open("rotunda.jpg")
        rotunda = ImageTk.PhotoImage(rot)
        label2 = Label(self, image=rotunda)
        label2.image = rotunda
        label2.place(x=40, y=160)

        minc = Image.open("mincol.jpg")
        mincol = ImageTk.PhotoImage(minc)
        label3 = Label(self, image=mincol)
        label3.image = mincol
        label3.place(x=170, y=50)
def main():
    root = Tk()
    root.geometry("300x280+300+300")
    app = Example()
    root.mainloop()
if __name__ == '__main__':
    main()
```

У цьому прикладі розташували три зображення за допомогою абсолютного позиціонування. Використали менеджер геометрії place.

```
from PIL import Image, ImageTk
```

Використовували `Image` та `ImageTk` з модуля `PIL` (Python Imaging Library).

```
style = Style()
style.configure("TFrame", background="#333")
```

За допомогою стилів змінили фон нашого вікна на темно-сірий.

```
bard = Image.open("bardejov.jpg")
bardejov = ImageTk.PhotoImage(bard)
```

Створили об'єкт зображення та об'єкт фото зображення зі збережених раніше зображень у поточній робочій директорії.

```
label1 = Label(self, image=bardejov)
```

Створили `Label` (ярлик) із зображенням. Дані ярлики можуть містити зображення і текст.

```
label1.image = bardejov
```

Нам потрібно зберегти посилання на зображення, щоб не втратити його, якщо збирач сміття (Garbage collector) його не закриє.

```
label1.place(x=20, y=20)
```

Ярлик розміщений у рамці за координатами x=20 та y=20.

Метод pack() розміщення віджетів по горизонталі та вертикалі.

Менеджер геометрії `pack()` упорядковує віджети у горизонтальні та вертикальні блоки. Макетом можна керувати за допомогою параметрів `fill`, `expand` та `side`.

Приклад створення кнопок у tkinter

У наступному прикладі ми розмістимо дві кнопки в правому нижньому кутку нашого вікна. Для цього ми скористаємося менеджером `pack`.

```
from tkinter import Tk, RIGHT, BOTH, RAISED
from tkinter.ttk import Frame, Button, Style
 class Example(Frame):
     def __init__(self):
        super().__init__()
        self.initUI()
    def initUI(self):
        self.master.title("Кнопки в tkinter")
        self.style = Style()
        self.style.theme_use("default")
        frame = Frame(self, relief=RAISED, borderwidth=1)
        frame.pack(fill=BOTH, expand=True)
        self.pack(fill=BOTH, expand=True)

        closeButton = Button(self, text=" Закрити")
        closeButton.pack(side=RIGHT, padx=5, pady=5)
        okButton = Button(self, text="Готово")
```

```
        okButton.pack(side=RIGHT)
def main():
    root = Tk()
    root.geometry("300x200+300+300")
    app = Example()
    root.mainloop()
if __name__ == '__main__':
    main()
```

Маємо дві рамки. Перша рамка – основна, а також друга – додаткова, яка розтягується в обидва боки та зсуває дві кнопки у нижню частину основної рамки. Кнопки знаходяться у горизонтальному контейнері та розміщені у її правій частині.

```
frame = Frame(self, relief=RAISED, borderwidth=1)
frame.pack(fill=BOTH, expand=True)
```

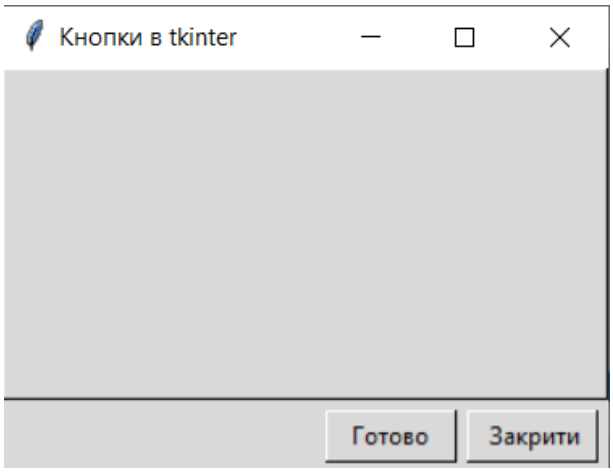
Ми створили ще один віджет `Frame`. Цей віджет займає практично весь простір вікна. Ми змінюємо межі рамки, щоб сама рамка була видна. За умовчанням вона пласка.

```
closeButton = Button(self, text="Закрити")
closeButton.pack(side=RIGHT, padx=5, pady=5)
```

Кнопка closeButton створена. Вона розташована у горизонтальному контейнері. Параметр side дозволяє помістити кнопку у правій частині горизонтальної смуги. Параметри padx та pady дозволяють встановити відступ між віджетами. Параметр `padx` встановлює простір між віджетами кнопки closeButton та правою межею кореневого вікна.

```
okButton.pack(side=RIGHT)
```

Кнопка `okButton` розміщена біля `closeButton` із встановленим відступом (`padding`) у 5 пікселів.



Створюємо програму для відгуків на Tkinter

Менеджер `pack` – це простий менеджер розмітки. Його можна використовувати для найпростіших завдань розмітки. Щоб створити складнішу розмітку, необхідно використовувати більше рамок, кожна з яких має власний менеджер розмітки.

```
from tkinter import Tk, Text, BOTH, X, N, LEFT
from tkinter.ttk import Frame, Label, Entry


class Example(Frame):

    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.master.title("Залишити відгук")
        self.pack(fill=BOTH, expand=True)

        frame1 = Frame(self)
        frame1.pack(fill=X)

        lbl1 = Label(frame1, text="Заголовок", width=10)
        lbl1.pack(side=LEFT, padx=5, pady=5)

        entry1 = Entry(frame1)
        entry1.pack(fill=X, padx=5, expand=True)

        frame2 = Frame(self)
        frame2.pack(fill=X)

        lbl2 = Label(frame2, text="Автор", width=10)
        lbl2.pack(side=LEFT, padx=5, pady=5)

        entry2 = Entry(frame2)
        entry2.pack(fill=X, padx=5, expand=True)

        frame3 = Frame(self)
        frame3.pack(fill=BOTH, expand=True)

        lbl3 = Label(frame3, text="Відгук", width=10)
        lbl3.pack(side=LEFT, anchor=N, padx=5, pady=5)

        txt = Text(frame3)
        txt.pack(fill=BOTH, pady=5, padx=5, expand=True)


def main():

    root = Tk()
```

```
    root.geometry("300x300+300+300")
    app = Example()
    root.mainloop()

if __name__ == '__main__':
    main()
```

На цьому прикладі видно, як можна створити складнішу розмітку з численними рамками та менеджерами `pack()`.

```
self.pack(fill=BOTH, expand=True)
```

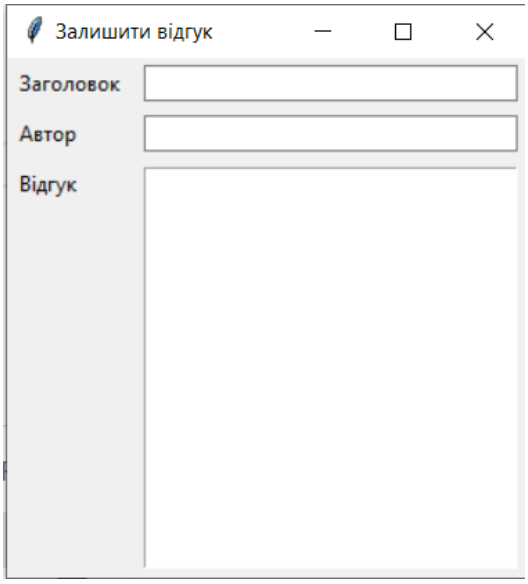
Перша рамка є базовою. На ній розташовуються всі інші рамки. Варто зазначити, що навіть при організації дочірніх віджетів у рамках ми керуємо ними на базовій рамці.

```
frame1 = Frame(self)
frame1.pack(fill=X)
lbl1 = Label(frame1, text="Заголовок", width=10)
lbl1.pack(side=LEFT, padx=5, pady=5)
entry1 = Entry(frame1)
entry1.pack(fill=X, padx=5, expand=True)
```

Перші два віджети розміщені на першій рамці. Поле для введення даних розтягнуте горизонтально з параметрами `fill` та `expand`.

```
frame3 = Frame(self)
frame3.pack(fill=BOTH, expand=True)

lbl3 = Label(frame3, text="Відгук", width=10)
lbl3.pack(side=LEFT, anchor=N, padx=5, pady=5)

txt = Text(frame3)
txt.pack(fill=BOTH, pady=5, padx=5, expand=True)
```

У третій рамці ми розмістили ярлик та віджет для введення тексту. Ярлик закріплений по північній стороні `anchor=N`, а віджет тексту займає решту простору.



Розмітка `grid()` у tkinter для створення калькулятора

Менеджер геометрії `grid()` tkinter використовується для створення сітки кнопок для калькулятора.

calculator.py

```
from tkinter import Tk, W, E
from tkinter.ttk import Frame, Button, Entry, Style


class Example(Frame):

    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.master.title("Калькулятор на Tkinter")

        Style().configure("TButton", padding=(0, 5, 0, 5),
                                            font='serif 10')

        self.columnconfigure(0, pad=3)
        self.columnconfigure(1, pad=3)
        self.columnconfigure(2, pad=3)
        self.columnconfigure(3, pad=3)

        self.rowconfigure(0, pad=3)
        self.rowconfigure(1, pad=3)
        self.rowconfigure(2, pad=3)
        self.rowconfigure(3, pad=3)
        self.rowconfigure(4, pad=3)

        entry = Entry(self)
```

```
            entry.grid(row=0, columnspan=4, sticky=W+E)
        cls = Button(self, text="Очистити")
            cls.grid(row=1, column=0)
            bck = Button(self, text="Видалити")
            bck.grid(row=1, column=1)
            lbl = Button(self)
            lbl.grid(row=1, column=2)
            clo = Button(self, text="Закрити")
            clo.grid(row=1, column=3)
            sev = Button(self, text="7")
            sev.grid(row=2, column=0)
            eig = Button(self, text="8")
            eig.grid(row=2, column=1)
            nin = Button(self, text="9")
            nin.grid(row=2, column=2)
            div = Button(self, text="/")
            div.grid(row=2, column=3)

            fou = Button(self, text="4")
            fou.grid(row=3, column=0)
            fiv = Button(self, text="5")
            fiv.grid(row=3, column=1)
            six = Button(self, text="6")
            six.grid(row=3, column=2)
            mul = Button(self, text="*")
            mul.grid(row=3, column=3)

            one = Button(self, text="1")
            one.grid(row=4, column=0)
            two = Button(self, text="2")
            two.grid(row=4, column=1)
            thr = Button(self, text="3")
            thr.grid(row=4, column=2)
            mns = Button(self, text="-")
            mns.grid(row=4, column=3)
             zer = Button(self, text="0")
            zer.grid(row=5, column=0)
            dot = Button(self, text=".")
            dot.grid(row=5, column=1)
            equ = Button(self, text="=")
            equ.grid(row=5, column=2)
            pls = Button(self, text="+")
            pls.grid(row=5, column=3)
            self.pack()
    def main():
        root = Tk()
        app = Example()
        root.mainloop()
    if __name__ == '__main__':
        main()
```

Менеджер `grid()` використовується для організації кнопок у контейнері рамки.

```
Style().configure("TButton", padding=(0, 5, 0, 5),
```

```
font='serif 10')
```

Ми налаштували віджет кнопки так, щоб відображався специфічний шрифт та застосовувався відступ (`padding`) у 3 пікселі.

```
self.columnconfigure(0, pad=3)
...
self.rowconfigure(0, pad=3)
```

Ми використовували методи `columnconfigure()` та `rowconfigure()`, щоб створити певний простір у сітці рядків та стовпців. Завдяки цьому кроку ми розділяємо кнопки певним порожнім простором.

```
entry = Entry(self)
entry.grid(row=0, columnspan=4, sticky=W+E)
```

Віджет поля введення – це місце, де відображатимуться цифри. Цей віджет розташований у першому ряду та охоплює всі чотири стовпці. Віджети можуть не займати весь простір, який виділяється клітинами у створеній сітці.

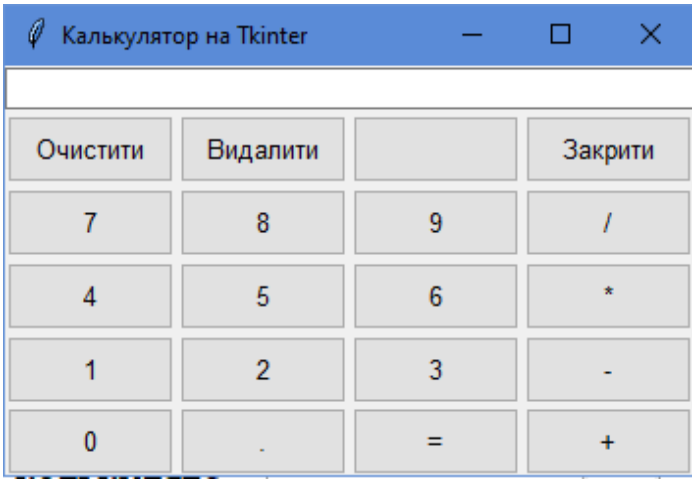
Параметр `sticky` розширює віджет у вказаному напрямку. У нашому випадку ми можемо переконатися, що наш віджет графи введення був розширений зліва направо W+E (схід–захід).

```
cls = Button(self, text="Очистити")
cls.grid(row=1, column=0)
```

Кнопка очищення встановлено у другому рядку та першому стовпці. Варто зазначити, що рядки та стовпці починаються з нуля.

```
self.pack()
```

Метод `pack()` показує віджет рамки та дає їй початковий розмір. Якщо додаткові параметри не вказуються, розмір буде таким, щоб усі дочірні віджети могли поміститися. Цей метод компонує віджет рамки у верхньому кореневому вікні, що також є контейнером. Менеджер `grid()` використовується для організації кнопок у віджеті рамки.



Приклад створення діалогового вікна в tkinter

Наступний приклад створює діалогове вікно за допомогою менеджера геометрії `grid`.

```
from tkinter import Tk, Text, BOTH, W, N, E, S
from tkinter.ttk import Frame, Button, Label, Style


class Example(Frame):

    def __init__(self):
        super().__init__()
        self.initUI()


    def initUI(self):

        self.master.title("Діалогове вікно в Tkinter")
        self.pack(fill=BOTH, expand=True)

        self.columnconfigure(1, weight=1)
        self.columnconfigure(3, pad=7)
        self.rowconfigure(3, weight=1)
        self.rowconfigure(5, pad=7)

        lbl = Label(self, text="Вікна ")
        lbl.grid(sticky=W, pady=4, padx=5)

        area = Text(self)
        area.grid(row=1, column=0, columnspan=2, rowspan=4,
                                        padx=5, sticky=E+W+S+N)

        abtn = Button(self, text="Активувати")
        abtn.grid(row=1, column=3)

        cbtn = Button(self, text="Закрити")
        cbtn.grid(row=2, column=3, pady=4)

        hbtn = Button(self, text="Допомога")
        hbtn.grid(row=5, column=0, padx=5)

        obtn = Button(self, text="Готово")
        obtn.grid(row=5, column=3)


def main():

    root = Tk()
    root.geometry("350x300+300+300")
    app = Example()
    root.mainloop()


if __name__ == '__main__':
    main()
```

У цьому прикладі ми використовували віджет ярлика, текстовий віджет та чотири кнопки.

```
self.columnconfigure(1, weight=1)
self.columnconfigure(3, pad=7)
self.rowconfigure(3, weight=1)
self.rowconfigure(5, pad=7)
```

Ми додали невеликий простір між віджетами у сітці. Параметр `weight` створює можливість розширення другого стовпця та четвертого ряду. У цьому рядку і стовпці знаходиться текстовий віджет, тому простір, що залишився, заповнює цей віджет.

```
lbl = Label(self, text="Вікна")
lbl.grid (sticky = W, pady = 4, padx = 5)
```

Віджет ярлика також створюється та поміщається у сітку. Якщо не вказуються ряд і стовпець, він займе перший ряд і стовпець. Ярлик закріплюється біля західної частини вікна `sticky=W` і має певні відступи навколо кордонів.

```
area = Text(self)
area.grid(row=1, column=0, columnspan=2, rowspan=4,
padx=5, sticky=E+W+S+N)
```

Створюється текстовий віджет і поміщається у другий рядок і перший стовпець. Він охоплює два стовпці та чотири рядки.

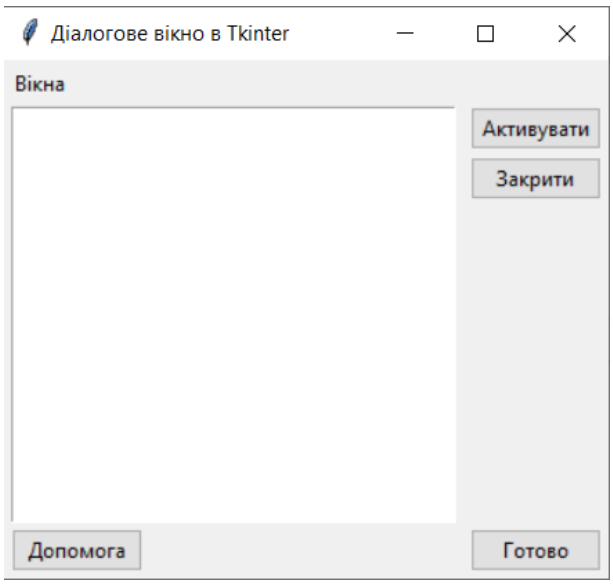
Між віджетом і лівим краєм кореневого вікна є простір у 4 пікселі. Також віджет закріплений біля всіх чотирьох сторін. Тому, коли вікно розширюється, віджети текстів збільшуються у всіх напрямках.

```
abtn = Button(self, text="Актив.")
abtn.grid(row=1, column=3)
cbtn = Button(self, text="Закрити")
cbtn.grid(row=2, column=3, pady=4)
```

Ці дві кнопки знаходяться біля текстового віджету.

```
hbtn = Button(self, text="Допомога")
hbtn.grid(row=5, column=0, padx=5)
obtn = Button (self, text = "Готово")
obtn.grid(row=5, column=3)
```

Ці дві кнопки знаходяться під текстовим віджетом. Кнопка «Допомога» розташована в першому стовпчику, а кнопка «Готово» в останньому стовпчику.



Питання для самоконтролю до теми 22

1. Яке призначення tkinter модуля?
2. Як створити головне графічне вікно?
3. Що таке віджет та як він створюється?
4. Як розміщуються віджети на графічній формі (вікні)?
5. Які віджети ви знаєте?

Завдання до теми 22

Змініть скрипт створення графічного вікна для калькулятора так, щоб кнопки «очистити», «видалити» та «закрити» розмістилися внизу графічного вікна калькулятора. Скрипт наведено у темі 22.

ВІДЖЕТИ (ГРАФІЧНІ ОБ'ЄКТИ) ТА ЇХ ВЛАСТИВОСТІ.

Розглянемо частину графічних об'єктів (віджет), які у бібліотеці tkinter: кнопки, поля для введення, мітки, прапорці, перемикачі і списки. Слід розуміти, що графічний інтерфейс користувача досить стандартний, і тому будь-які подібні бібліотеки-модулі (в тому числі і tkinter) містять приблизно однакові віджети.

Кожен клас віджет має певні властивості, значення яких можна задавати під час їх створення, а також програмувати їх зміну при дії користувача та в результаті виконання програми.

Кнопки

Об'єкт-кнопка створюється викликом класу Button tkinter модуля. При цьому обов'язковим аргументом є лише батьківський віджет (наприклад вікно верхнього рівня). Інші властивості можуть вказуватися під час створення кнопки або задаватися (змінюватися) пізніше. Синтаксис:

```
змінна = Button (родить_віджет, [властивість = значення, … ….])
```

Кнопка має багато властивостей, у прикладі нижче вказані лише деякі з них.

```
from tkinter import *

root = Tk()

but = Button(root,
text="Це кнопка", #напис на кнопці
width=30,height=5, #ширина та висота
bg="white",fg="blue") #колір фону та написи

but.pack()
root.mainloop()
```

`bg` та `fg` – це скорочення від `background` (фон) та `foreground` (передній план). Ширина та висота вимірюються в знакомісцях (кількість символів).

Мітки

Мітки (або написи) — це досить прості віджети, що містять рядок (або кілька рядків) тексту та службовці, в основному, для інформування користувача.

```
lab = Label(root, text="Це мітка!\n3 2 рядків.", font="Arial 18")
```

Однорядкове текстове поле

Таке поле створюється викликом класу модуля `Entry` `tkinter`. Користувач може ввести лише один рядок тексту.

```
ent = Entry (root, width = 20, bd = 3)
```

`bd`- Це скорочення від межі ширини (ширина кордону).

Елемент `Entry` є полем для введення тексту. Конструктор `Entry` приймає такі параметри:

```
Entry(master, options)
```

Де `master` – посилання на батьківське вікно, а `options` – набір наступних параметрів:

- `bg`: фоновий колір
- `bd`: товщина кордону
- `cursor`: курсор покажчика миші при наведенні на текстове поле
- `fg`: колір тексту
- `font`: шрифт тексту
- `justify`: встановлює вирівнювання тексту Значення `LEFT` вирівнює текст по лівому краю, `CENTER` – по центру, `RIGHT` – по правому краю
- `relief`: визначає тип кордону, за замовчуванням значення `FLAT`
- `selectbackground`: фоновий колір виділеного шматка тексту.
- `selectforeground`: колір виділеного тексту
- `show`: задає маску для символів, що вводяться.
- `state`: стан елемента може приймати значення `NORMAL` (за замовчуванням) і `DISABLED`
- `textvariable`: встановлює прив'язку до елемента `StringVar`
- `width`: ширина елемента

Визначимо елемент `Entry` та за натисканням на кнопку виведемо його текст в окреме вікно з повідомленням:

```
from tkinter import *
from tkinter import messagebox
def show_message():
   messagebox.showinfo("GUI Python", message.get())

root =Tk()
root.title("GUI на Python")
root.geometry("300x250")

message =StringVar()

message_entry =Entry(textvariable=message)
message_entry.place(relx=.5, rely=.1, anchor="c")

message_button =Button(text="Click Me",  command=show_message)
message_button.place(relx=.5, rely=.5, anchor="c")

root.mainloop()
```

Для виведення повідомлення тут застосовується додатковий модуль `messagebox`, що містить функцію `showinfo()`, яка і виводить введений у текстове поле текст. Для отримання введеного тексту використовується комп'ютер `StringVar`.

Тепер створимо складніший приклад із формою введення:

```
from tkinter import *
```

```
fromt kinter import messagebox


def display_full_name():
   messagebox.showinfo("GUI Python", name.get() +
                                          " " + surname.get())

root =Tk()
root.title("GUI на Python")

name =StringVar()
surname =StringVar()

name_label =Label(text="Введіть ім'я:")
surname_label =Label(text="Введіть прізвище:")

name_label.grid(row=0, column=0, sticky="w")
surname_label.grid(row=1, column=0, sticky="w")

name_entry =Entry(textvariable=name)
surname_entry =Entry(textvariable=surname)

name_entry.grid(row=0,column=1, padx=5, pady=5)
surname_entry.grid(row=1,column=1, padx=5, pady=5)


message_button = Button(text="Click Me",
command=display_full_name)
message_button.grid(row=2,column=1, padx=5, pady=5,
sticky="e")

root.mainloop()
```

Методи `Entry`

Елемент `Entry` має низку методів. Основні з них:

- `insert(index, str)`: вставляє в текстове поле рядок за певним індексом
- `get()`: повертає введений текстове поле текст
- `delete(first, last=None)`: видаляє символ за індексом `first`. Якщо вказано параметр `last`, видалення проводиться до індексу `last`. Щоб видалити до кінця, в якості другого параметра можна використовувати значення `END`.

Використовуємо методи у програмі:

```
from tkinter import *
from tkinter import messagebox
```

```
Def clear():
   name_entry.delete(0, END)
   surname_entry.delete(0, END)

def display():
   messagebox.showinfo("GUI Python", name_entry.get() +
                              " " +  surname_entry.get())

root =Tk()
root.title("GUI на Python")

name_label =Label(text="Введіть ім'я:")
surname_label =Label(text="Введіть прізвище:")

name_label.grid(row=0, column=0, sticky="w")
surname_label.grid(row=1, column=0, sticky="w")

name_entry =Entry()
surname_entry =Entry()

name_entry.grid(row=0,column=1, padx=5, pady=5)
surname_entry.grid(row=1,column=1, padx=5, pady=5)

# Вставка початкових даних
name_entry.insert(0, "Tom")
surname_entry.insert(0, "Soyer")

display_button =Button(text="Display", command=display)
clear_button =Button(text="Clear", command=clear)

display_button.grid(row=2, column=0, padx=5, pady=5, sticky="e")
clear_button.grid(row=2, column=1, padx=5, pady=5, sticky="e")

root.mainloop()
```

При запуску програми відразу в обидва текстові поля додається текст за замовчуванням:

```
name_entry.insert(0, "Tom")
surname_entry.insert(0, "Soyer")

name_entry.insert("end", "Tom")
```

А так можна вставити текст «наприкінці поля»

Кнопка `Clear` очищає обидва поля, викликаючи метод `delete`:

```
def clear():
   name_entry.delete(0, END)
   surname_entry.delete(0, END)
```

Друга кнопка, використовуючи метод `get`, отримує введений текст:

```
def display():
   messagebox.showinfo("GUI Python", name_entry.get() +
                                        " " + surname_entry.get())
```

Причому, як видно з прикладу, нам необов'язково звертатися до тексту в Entry через змінні типу `StringVar`, ми можемо зробити це безпосередньо через метод `get`

Багаторядкове текстове поле

Text призначений для надання користувачеві можливості введення не одного рядка тексту, а значно більше.

```
tex = Text (root, width = 40, font="Verdana 12", wrap = WORD)
```

Остання властивість (`wrap`) в залежності від свого значення дозволяє переносити текст, що вводиться користувачем або за символами, або за словами, або взагалі не переносити, доки користувач не натисне `Enter`.

Радіокнопки (перемикачі)

Об'єкт-радіокнопка ніколи не використовується по одному. Їх використовують групами, при цьому в одній групі може бути включена лише одна кнопка.

```
var=IntVar()
var.set(1)
rad0 = Radiobutton(root,text="Перша",
variable=var,value=0)
rad1 = Radiobutton(root,text="Друга",
variable=var,value=1)
rad2 = Radiobutton(root,text="Третя",
variable=var,value=2)
```

Одна група визначає значення однієї змінної, тобто якщо в прикладі буде обрана радіокнопка `rad2`, то змінної буде `var` буде 2.

Спочатку також потрібно встановити значення змінної (вираз `var.set(1)` задає значення змінної `var` рівне 1).

Прапорці

Об'єкт `checkbutton` призначений для вибору не взаємовиключних пунктів у вікні (у групі можна активувати один, два або більше прапорців або один). На відміну від радіокнопок, значення кожного прапорця прив'язується до своєї змінної, значення якої визначається опціями onvalue (включено) та `offvalue` (вимкнено) в описі прапорця.

```
c1 = IntVar()
c2 = IntVar()
che1 = Checkbutton(root,text="Перший прапорець",
variable=c1,onvalue=1,offvalue=0)
```

```
che2 = Checkbutton(root,text="Другий прапорець",
variable=c2,onvalue=2,offvalue=0)
```

Списки

Виклик класу `Listbox` створює об'єкт, в якому користувач може вибрати один або кілька пунктів залежно від значення опції `selectmode`. У прикладі нижче значення `SINGLE` дозволяє вибирати лише один пункт зі списку.

```
r = ['Linux','Python','Tk','Tkinter']
lis = Listbox(root,selectmode=SINGLE,height=4)
for i in r:
   lis.insert(END,i)
```

Спочатку список (`Listbox`) порожній. За допомогою циклу `for` до нього додаються пункти зі списку (тип даних) `r`. Додавання відбувається за допомогою спеціального методу класу `Listbox – insert`. Даний метод приймає два параметри: куди додати та що додати.

Більшість методів різних віджет ми розглянемо під час вивчення даного курсу.

Віджети (графічні об'єкти) та їх властивості

Продовжимо розглядати графічні об'єкти (віджети), які у бібліотеці tkinter. Це будуть рамка (`frame`), шкала (`scale`), смуга прокручування (`scrollbar`), вікно верхнього рівня (`toplevel`).

`Frame` (рамка)

Як з'ясується пізніше, рамки (фрейми) є гарним інструментом для організації інших віджет у групи всередині вікна, а також оформлення.

```
from tkinter import *

root = Tk()

fra1 = frame(root,width=500,height=100,bg="darkred")
fra2 = frame(root,width=300,height=200,bg="green",bd=20)
fra3 = Frame(root,width=500,height=150,bg="darkblue")

fra1.pack()
fra2.pack()
fra3.pack()

root.mainloop()
```

Цей скрипт створює три кадри різного розміру. Властивість `bd` (скорочення від `boderwidth`) визначає відстані від краю рамки до віджетів, що у неї віджетів (якщо вони є).

На фреймах можна розміщувати віджети як на основному вікні (`root`). Тут текстове поле знаходиться на рамці `fra2`.

```
ent1 = Entry(fra2,width=20)
```

```
ent1.pack()
```

Scale (шкала)

Призначення шкали – це надання користувачеві вибору певного значення з певного діапазону. Зовні шкала є горизонтальною або вертикальною смугою з розміткою, по якій користувач може пересувати движок, здійснюючи тим самим вибір значення.

```
sca1 = Scale(fra3,orient=HORIZONTAL,length=300,
from_=0,to=100,tickinterval=10,resolution=5)
sca2 = Scale(root,orient=VERTICAL,length=400,
from_=1,to=2,tickinterval=0.1,resolution=0.1)
```

Властивості:

- `orient` - визначає напрямок шкали;
- `length`– довжина шкали в пікселях;
- `from_` і `to` – з якого значення шкала починається і яким закінчується (тобто діапазон значень);
- `tickinterval` – інтервал, через який відображаються позначки для шкали;
- `resolution`– мінімальна довжина відрізка, на яку користувач може пересунути двигун.

Scrollbar (смуга прокручування)

Цей віджет дозволяє прокручувати вміст іншого віджету (наприклад, текстового поля або списку). Прокручування може бути як по горизонталі, так і по вертикалі.

```
from tkinter import *
root = Tk()
tx = Text(root,width=40,height=3,font='14')
scr = Scrollbar(root,command=tx.yview)
tx.configure(yscrollcommand=scr.set)

tx.grid(row=0,column=0)
scr.grid(row=0,column=1)
root.mainloop()
```

У прикладі спочатку створюється текстове поле (`tx`), потім смуга прокручування (`scr`), яка прив'язується за допомогою опції command до поля tx вертикальної осі (`yview`). Далі поле tx змінюється (конфігурується) за допомогою методу configure: встановлюється значення опції `yscrollcommand`.

Тут використовується незнайомий нам поки ще метод `grid`, що є іншим способом розташування віджет на вікні.

`Toplevel` (вікно верхнього рівня)

За допомогою класу `Toplevel` створюються дочірні вікна, на яких також можуть розташовуватися віджети. Слід зазначити, що при закритті головного вікна (або батьківського) вікно `Toplevel` також закривається. З іншого боку, закриття дочірнього вікна не призводить до закриття головного.

```
win = Toplevel(root, relief = SUNKEN, bd = 10, bg = "lightblue")
win.title("Дочірнє вікно")
win.minsize(width=400,height=200)
```

Метод `title` визначає заголовок вікна. Метод minsize конфігурує мінімальний розмір вікна (є метод `maxsize`, що визначає максимальний розмір вікна). Якщо значення аргументів `minsize` буде таким самим, як у maxsize, то користувач не зможе змінювати розміри вікна.

Питання для самоконтролю до теми 23

1. Які характеристики віджета «кнопка» ви знаєте?
2. Яке призначення віджету «Однорядкове текстове поле» ви знаєте?
3. Які методи віджету Entry ви знаєте?
4. Як можна використовувати віджети «радіокнопка» та «прапорці»?
5. Призначення віджету Frame (рамка) та його властивості?

Завдання до теми 23

Розробити GUI інтерфейс для програми побудови заданих користувачем графіків функцій y=f(x).

Інтерфейс повинен містити вікна для введення:

- значень інтервалу змінюючи значення змінної x [a, b];
- введення двох функціональних залежностей y1=f1(x) та y2=f2(x) у вигляді текстового рядка формул;
- кнопку для очищення малюнка графіків;
- кнопку для побудови графіків;
- кнопку для завершення програми.

24. ПРОГРАМУВАННЯ ПОДІЙ

Метод bind модуля tkinter

Програми з графічним інтерфейсом користувача (GUI) повинні не просто красиво відображатися на екрані, але й виконувати будь-які дії, реалізуючи потреби користувача. Розглянемо, як додати йому функціональність, тобто можливість здійснювати за його допомогою ті чи інші дії.

На відміну від консольних додатків, які зазвичай виконуються при мінімальних зовнішніх впливах, графічний додаток зазвичай чекає на будь-які зовнішні впливи (клацань кнопкою миші, натискання клавіш на клавіатурі, зміни віджетів) і потім виконує закладену програмістом дію.

З такого принципу роботи можна вивести наступну схему налаштування функціональності GUI:

на віджет щось «впливає» «зовні»? – виконується якась функція (дія). Зовнішній вплив на графічний компонент називається подією.

Події досить багато (основний їх перелік ми розглянемо у наступному). Зараз будемо використовувати лише два види подій: клацання лівою кнопкою миші та натискання клавіші Enter.

Одним із способів зв'язування віджету, події та функції (те, що має відбуватися після події) є використання методу bind. Синтаксис зв'язування представлений нижче:

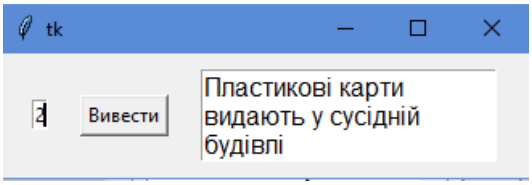
```
widget.bind(event, function)
```

Розглянемо різні приклади додавання функціональності GUI.

Приклад 1.

```
def output(event):
   s = ent.get()
   if s == "1":
     tex.delete(1.0,END)
     tex.insert(END,"Обслуговування клієнтів на
                                          другому поверсі")
   elif s == "2":
     tex.delete(1.0,END)
     tex.insert(END,"Пластикові карти видають у
                                    сусідній будівлі")
   else:
     tex.delete(1.0,END)
     tex.insert(END,"Введіть 1 або 2 у поле зліва")
from tkinter import *
root = Tk()
ent = Entry(root,width=1)          ## 16
but = Button(root,text="Вивести") ## 17
tex =Text(root,width=20,height=3,font="12",wrap=WORD)
ent.grid(row=0,column=0,padx=20)  ## 22
but.grid(row=0,column=1)
tex.grid(row=0,column=2,padx=20,pady=10)
```

```
but.bind("<Button-1>",output)     ## 24
root.mainloop()
```



Розглянемо код, починаючи з 16-го рядка.

У рядках 16-18 створюються три віджети: однорядкове текстове поле, кнопка та багаторядкове текстове поле. У першому полі користувач повинен щось ввести, потім натиснути кнопку і отримати відповідь у другому полі.

У рядках 20-22 використовується менеджер grid для розміщення віджетів. Властивості padx та pady визначають кількість пікселів від віджету до краю рамки (або комірки) по осях x та y відповідно.

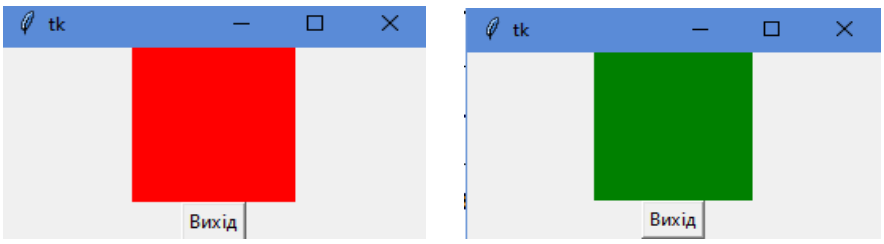
У стоку 24 саме відбувається зв'язування кнопки з подією натискання лівої кнопки миші і функцією output. Всі ці три компоненти (віджет, подія та функція) зв'язуються за допомогою методу bind. В даному випадку при натисканні лівою кнопкою миші по кнопці but буде викликана функція output.

Отже, якщо раптом користувач клацне лівою кнопкою миші по кнопці, виконається функція output (ні в якому іншому випадку вона виконуватися не буде). Ця функція (рядки 1-11) виводить інформацію у друге текстове поле. Яку саме інформацію залежить від того, що користувач ввів у перше текстове поле. Як аргумент функції передається подія (у разі).

Усередині гілок if-elif-else використовуються методи delete та insert. Перший видаляє символи з текстового поля, другий — вставляє. 1.0 – позначає перший рядок, перший символ (нумерація символів починається з нуля).

Приклад 2.

```
li = ["red","green"]
def color(event):
   fra.configure(bg=li[0])
   li[0],li[1] = li[1],li[0]
def outgo(event):
   root.destroy()
from tkinter import *
root = Tk()
fra = Frame (root, width = 100, height = 100)  # 12 & 13
but = Button(root,text="Вихід")
fra.pack()
but.pack()
root.bind("<Return>",color)      # 18
but.bind("<Button-1>",outgo)     # 19
root.mainloop()
```



Тут створюються два віджети (рядки 12, 13): кадр і кнопка.

Програма реагує на дві події: натискання клавіші Enter в межах головного вікна (рядок 18) та натискання лівою кнопкою миші по кнопці but (рядок 19). У першому випадку викликається функція color, у другому – outgo.

Функція color змінює колір фону (bg) кадру (fra) за допомогою методу configure, який призначений для зміни значення властивостей віджетів у процесі скрипту. Як значення опції bg підставляється перший елемент списку. Потім у списку два елементи змінюються місцями, щоб при наступному натисканні Enter колір кадру знову змінився.

У функції outgo викликається метод destroy по відношенню до головного вікна. Цей метод призначений для «руйнування» віджету (вікно закриється).

Програмування подій у tkinter

Зазвичай, щоб графічний додаток щось зробило, має статися якесь подія, т.е. вплив на GUI з-за.

Типи подій

Можна виділити три основні типи подій: миші, натискання клавіш на клавіатурі, а також події, що виникають в результаті зміни інших графічних об'єктів.

Спосіб запису

При виклику методу bind подія передається як перший аргумент.



Назва події полягає в лапки, а також знаки < і >. Подія описується за допомогою зарезервованих послідовностей ключових слів.

Події, що виробляються мишею

- `<Button-1>`- клацання лівою кнопкою миші
- `<Button-2>`- клацання середньою кнопкою миші
- `<Button-3>`- клацання правою кнопкою миші
- `<Double-Button-1>`- подвійний клік лівою кнопкою миші
- `<Motion>`- рух миші.

Приклад:

```
from tkinter import *
def b1(event):
   root.title("Ліва кнопка миші")
def b3(event):
   root.title("Права кнопка миші")
def move(event):
   root.title("Рух мишею")
root = Tk()
root.minsize(width = 500, height = 400)
root.bind('<Button-1>',b1)
root.bind('<Button-3>',b3)
root.bind('<Motion>',move)
root.mainloop()
```

У цій програмі змінюється напис у заголовку головного вікна залежно від того, що рухається миша, клацають лівою або правою кнопкою миші.

Подія (`Event`) – це один із об'єктів tkinter. Події мають атрибути, як і багато інших об'єктів. У прикладі функції move витягуються значення атрибутів x і y об'єкта `event`, в яких зберігаються координати розташування курсору миші в межах віджету, по відношенню до якого була згенерована подія. У разі віджетом є головне вікно, а подією – `<Motion>`, т. е. переміщення миші.

У програмі нижче виводиться інформація про примірник `event` та деякі його властивості. Усі атрибути можна переглянути за допомогою команди `dir(event)`. У різних подій вони ті самі, змінюються лише значення. Для тих чи інших подій частина атрибутів немає сенсу, такі властивості мають значення за умовчанням.

У прикладі хоча обробляється подія *натискання клавіші клавіатури*, поля x, y, `x_root`, `y_root` зберігаються координати положення на екрані курсора миші.

```
from tkinter import *

def event_info(event):
    print(type(event))
    print(event)
    print(event.time)
    print(event.x_root)
    print(event.y_root)
root=Tk()
root.bind('a',event_info)
root.mainloop()
```

Приклад виконання програми:

```
<class 'tkinter.Event'>
<KeyPress event state=Mod2 keysym=a keycode=38
char='a' x=9 y=7>
8379853
37
```

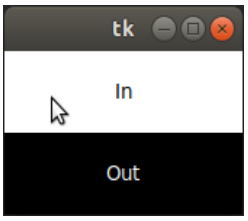
92

Для подій з клавіатури можна записати буквені клавіші без кутових дужок (наприклад, 'a').

Для неалфавітних кнопок існують спеціальні зарезервовані слова. Наприклад, <Return> – натискання клавіші Enter, <space> – пробіл. (Зауважимо, що є подія <Enter>, яка не має відношення до натискання клавіші Enter, а відбувається, коли курсор заходить у межі віджету.)

Розглянемо програму:

```
from tkinter import *

def enter_leave(event):
    if event.type == '7':
        event.widget['text'] = 'In'
    elif event.type == '8':
        event.widget['text'] = 'Out'
root = Tk()
lab1 = Label(width=20, height=3, bg='white')
lab1.pack()
lab1.bind('<Enter>', enter_leave)
lab1.bind('<Leave>', enter_leave)
lab2 = Label(width=20, height=3, bg='black',
                                          fg='white')
lab2.pack()
lab2.bind('<Enter>', enter_leave)
lab2.bind('<Leave>', enter_leave)
root.mainloop()
```



У ній дві мітки використовують одну й ту саму функцію, і кожна мітка використовує цю функцію для обробки двох різних подій: введення курсору в межі віджету та виведення у межі.

Функція, залежно від того, стосовно якого віджету було зафіксовано подію, змінює властивості тільки цього віджету. Як змінює, залежить від події, що сталася.

Властивість `event.widget` містить посилання на віджет, що згенерував подію. Властивість event.type описує, що це була подія. Кожна подія має ім'я та номер.

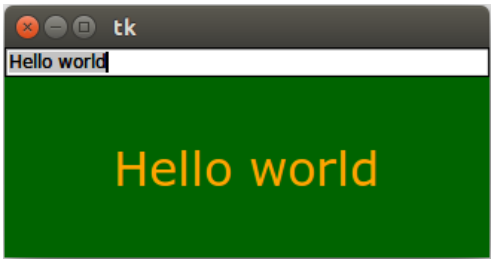
За допомогою виразу `print(repr(event.type))` можна переглянути його повний опис.

При цьому на одних платформах `str(event.type)` повертає ім'я події (наприклад, `Enter`), на інших – рядкове представлення номера події (наприклад, 7).

Повернімося до подій клавіатури. Поєднання клавіш пишуться через тире. У разі використання так званого модифікатора він вказується першим, деталі на третьому місці.

Наприклад, `<Shift-Up>` – одночасне натискання клавіш Shift та стрілки вгору, `<Control-B1-Motion>` – рух мишею із затиснутою лівою кнопкою та клавішею Ctrl.

```
from tkinter import *
def exit_win(event):
    root.destroy()
def to_label(event):
    t = ent.get()
    lbl.configure(text=t)
def select_all(event):
    def select_all2(widget):
        widget.selection_range(0, END)
        widget.icursor(END)   # курсор в кінец
    root.after(10, select_all2, event.widget)
root = Tk()
ent = Entry(width=40)
ent.focus_set()
ent.pack()
lbl = Label(height=3, fg='orange',
                  bg='darkgreen', font="Verdana 24")
lbl.pack(fill=X)
ent.bind('<Return>', to_label)
ent.bind('<Control-a>', select_all)
root.bind('<Control-q>', exit_win)
root.mainloop()
```



Тут поєднання клавіш Ctrl+a виділяє текст у полі. Без `root.after()` виділення не працює. Метод `after` виконує функцію, вказану у другому аргументі, через проміжок часу, вказаний у першому аргументі. У третьому аргументі передається значення атрибута widget об'єкта `event`. У цьому випадку їм буде поле `ent`. Саме воно буде передано як аргумент у функцію `select_all2` та присвоєно параметру `widget`.

Події, що виробляються за допомогою клавіатури

– Літерні клавіші можна записувати без кутових дужок (наприклад, 'L').
– Для неалфавітних клавіш існують спеціальні зарезервовані слова * <Return> – натискання клавіші Enter; * <space> – пробіл; * та інше.
– Поєднання клавіш пишуться через тире. Наприклад: * <Control-Shift> – одночасне натискання клавіш Ctrl та Shift.

Приклад:

```
from tkinter import *
def exit_(event):
   root.destroy()
def caption(event):
   t = ent.get()
   lbl.configure(text = t)
root = Tk()
ent = Entry (root, width = 40)
lbl = Label (root, width = 80)
ent.pack()
lbl.pack()
ent.bind('<Return>',caption)
root.bind('<Control-z>',exit_)
root.mainloop()
```

При натисканні клавіші Enter в межах текстового рядка (`ent`) викликається функція caption, яка поміщає символи з текстового рядка (`ent`) до мітки (`lbl`). Натискання комбінації клавіш Ctrl+z призводить до закриття головного вікна.

Питання для самоконтролю до теми 24

1. Що таке подія. Які події обробляються tkinter?
2. Яке призначення методу `bind` модуля tkinter?
3. Які типи подій модуля tkinter ви знаєте?
4. Які події виконуються маніпулятором миша?
5. Що містить властивість event.widget?

Завдання до теми 24

Для скрипту побудови графічного вікна калькулятора (див. тему 22) розробити функції обробки подій натискання функціональних кнопок.

25. ЗМІННІ tkinter

Бібліотека tkinter містить спеціальні класи, об'єкти яких виконують роль змінних для зберігання значень про стан різних віджетів. Зміна значення такої змінної веде до зміни властивості віджету, і навпаки: зміна властивості віджета змінює значення асоційованої змінної.

Існує кілька класів tkinter, призначених для обробки даних різних типів.

1. `StringVar()` – для рядків;
2. `IntVar()` – цілих чисел;
3. `DoubleVar()` – дробових чисел;
4. `BooleanVar()` – для обробки булевих значень (true та false).

Приклад 1.

Раніше ми вже використовували змінну- об'єкт типу `IntVar()` при створенні групи радіокнопок:

```
var=IntVar()
var.set(1)
rad0 = \
  Radiobutton(root,text="Перша",variable=var,value=0)
rad1 = \
  Radiobutton(root,text="Друга",variable=var,value=1)
rad2 = \
    Radiobutton(root,text="Третя",variable=var,value=2)
```

Тут створюється об'єкт класу `IntVar` та зв'язується зі змінною `var`. За допомогою методу `set` встановлюється початкове значення 1.

Три радіокнопки відносяться до однієї групи: про це свідчить однакове значення опції (властивості) `variable`.

Variable призначена для зв'язування змінної tkinter із радіокнопкою. Опція value визначає значення, яке буде передано змінною, якщо ця кнопка буде в змозі "ввімкнено". Якщо у процесі виконання скрипту значення змінної var буде змінено, це позначиться групі кнопок. Наприклад, це робиться в другому рядку коду: увімкнена кнопка `rad1`.

Якщо метод `set` дозволяє встановлювати значення змінних, то метод `get`, навпаки, дозволяє отримувати (пізнавати) значення подальшого їх використання.

```
def display(event):
   v = var.get()
   if v == 0:
      print ("Увімкнена перша кнопка")
   elif v == 1:
     print ("Включена друга кнопка")
   elif v == 2:
     print ("Увімкнена третя кнопка")
```

```
but = Button(root,text="Отримати значення")
but.bind('<Button-1>',display)
```

При виклику функції display в змінну v “записується“ значення, пов'язане з змінною var. Щоб отримати значення змінної var, використовується метод get (другий рядок коду).

Приклад 2.

Дещо складніше справа з прапорцями. Оскільки стани прапорців незалежні один одного, то для кожного має бути введена власна асоційована змінна-об'єкт.

```
from tkinter import *

root = Tk()

var0=StringVar() # значення кожного прапорця ...
var1=StringVar() # ... зберігається у власній змінній
var2=StringVar()
# якщо прапорець встановлений, то асоційовану змінну ...
# ... (var0, var1 або var2) заноситься значення
#                  onvalue, ...
# ...якщо прапорець знято, то - offvalue.
ch0 = Checkbutton(root,text="Коло",variable=var0,
onvalue="circle",offvalue="-")
ch1 = Checkbutton(root,text="Квадрат",variable=var1,
onvalue="square",offvalue="-")
ch2 = Checkbutton(root,text="Трикутник",variable=var2,
onvalue="triangle",offvalue="-")


lis = Listbox(root,height=3)
def result(event):
  v0 = var0.get()
  v1 = var1.get()
  v2 = var2.get()
  l = [v0,v1,v2] # значення змінних заносяться до списку
  lis.delete(0,2) # попередній вміст видаляється з
                                      #       Listbox
  for v in l: # вміст списку l послідовно
    lis.insert(END,v) # ...вставляється в Listbox

but = Button(root,text="Отримати значення")
but.bind('<Button-1>',result)

ch0.deselect()# "за замовчуванням" прапорці зняті
ch1.deselect()
ch2.deselect()

ch0.pack()
ch1.pack()
ch2.pack()
but.pack()
lis.pack()
```

```
root.mainloop()
```

Приклад 3.

Крім властивості (опції) variable, що зв'язує віджет зі змінною-об'єктом tkinter (`IntVar`, `StringVar` та інше), у багатьох віджет існує опція `textvariable`, яка визначає текст-вміст або текст-напис віджету. Незважаючи на те, що «текстова властивість» може бути встановлена для віджету і змінена в процесі виконання коду без використання асоційованих змінних, іноді такий спосіб зміни виявляється зручнішим.

```
from tkinter import *
root = Tk()
v = StringVar()
ent1 = Entry (root, textvariable=v, bg="black", fg = "white")

ent2 = Entry(root, textvariable = v)
ent1.pack()
ent2.pack()
root.mainloop()
```

Тут вміст одного текстового поля негайно відображається в іншому, т.к. обидва поля прив'язані до однієї і тієї ж змінної v.

Питання для самоконтролю до теми 25

1. Що таке "змінні Tkinter"?
2. Які класи "змінних Tkinter" ви знаєте?
3. Як створити об'єкт класу IntVar?
4. Скільки змінних Tkinter потрібно створити для віджету радіокнопка?
5. Скільки змінних Tkinter потрібно створити для віджета прапорці?

Завдання до теми 25

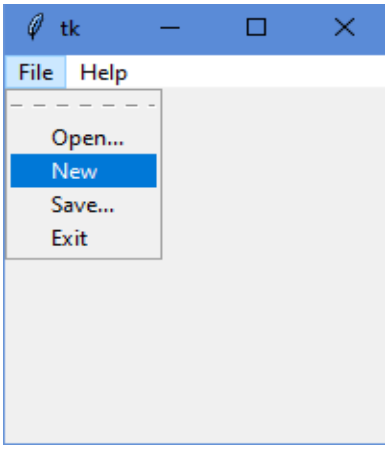
Для віконного інтерфейсу програми побудови заданих користувачем графіків функцій y=f(x) реалізуйте отримання параметрів, що задаються через змінні Tkinter (дивись завдання до теми 22).

26. Об'єкт меню в GUI.

Меню – це об'єкт, який присутній у багатьох програмах користувача. Знаходиться воно під рядком заголовка і являє собою списки, що випадають під словами; кожен такий список може містити інший вкладений список. Кожен пункт списку є командою, яка запускає будь-яку дію або відкриває діалогове вікно.

Створення меню в Tkinter

```
from tkinter import *
root = Tk()
m = Menu(root) # створюється об'єкт Меню
               # на головному вікні
root.config(menu=m) # вікно конфігурується
                    # із зазначенням меню для нього
fm = Menu(m) # створюється пункт меню з
             # розміщенням на основному меню (m)
m.add_cascade(label="File",menu=fm) #пункту
                   # розташовується на основному меню (m)
fm.add_command(label="Open...") #формується список
                                #   команд пункту меню

fm.add_command(label="New")
fm.add_command(label="Save...")
fm.add_command(label="Exit")

hm = Menu(m) #другий пункт меню
m.add_cascade(label="Help",menu=hm)
hm.add_command(label="Help")
hm.add_command(label="About")

root.mainloop()
```



Метод `add_cascade` додає новий пункт до меню, який вказується як значення опції menu.

Метод `add_command` додає нову команду до пункту меню. Одна з опцій даного методу (у прикладі вище її поки що немає) – `command` – пов'язує цю команду з функцією-обробником.

Можна створити вкладене меню. Для цього створюється ще одне меню та за допомогою `add_cascade` прив'язати до батьківського пункту.

```
nfm = Menu(fm)
fm.add_cascade(label="Import",menu=nfm)
nfm.add_command(label="Image")
nfm.add_command(label="Text")
```

Прив'язка функцій до меню

Кожна команда меню зазвичай має бути пов'язана зі своєю функцією, яка виконує ті чи інші дії (вирази). Зв'язок відбувається з допомогою опції command методу add_command. Функція оброблювач до цього має бути визначена.

Для прикладу вище наведено виправлені рядки додавання команд “About”, “New” і “Exit”, а також функції, що викликаються, коли користувач клацає лівою кнопкою миші по відповідним пунктам підменю.

```
def new_win():
   win = Toplevel(root)
def close_win():
   root.destroy()
def about():
   win = Toplevel(root)
lab = Label(win,text="Це просто програма-тест \n меню в Tkinter")
lab.pack()
fm.add_command(label="New",command=new_win)
fm.add_command(label="Exit",command=close_win)
hm.add_command(label="About",command=about)
```

Вправа – приклад

Напишемо програму з меню, що містить два пункти: Color і Size. Пункт Color повинен містити три команди (Red, Green та Blue), що змінюють колір рамки на головному вікні. Пункт Size повинен містити дві команди (500x500 та 700x400), що змінюють розмір рамки.

Зразкове рішення:

```
from tkinter import *
root = Tk()
def colorR():
     fra.config(bg="Red")
def colorG():
     fra.config(bg="Green")
def colorB():
     fra.config(bg="Blue")
def square():
     fra.config(width=500)
     fra.config(height=500)
```

```
def rectangle():
     fra.config(width=700)
     fra.config(height=400)
fra = Frame(root,width=300,height=100,bg="Black")
fra.pack()
m = Menu(root)
root.config(menu=m)

cm = Menu(m)
m.add_cascade(label="Color",menu=cm)
cm.add_command(label="Red",command=colorR)
cm.add_command(label="Green",command=colorG)
cm.add_command(label="Blue",command=colorB)

sm = Menu(m)
m.add_cascade(label="Size",menu=sm)
sm.add_command(label="500x500",command=square)
sm.add_command(label="700x400",command=rectangle)

root.mainloop()
```



Питання для самоконтролю до теми 26

1. Що таке об'єкт «меню» та його призначення?
2. Як створити об'єкт меню?
3. Як створити вкладене меню?
4. Як прив'язати функції користувача до пунктів меню?
5. Як формується список команд пункту меню?

Завдання до теми 26

Напишіть програму з меню, що містить пункти: Color, Size та End. Пункт Color повинен містити команди (Red, Green та Blue), що змінюють колір рамки на головному вікні. Пункт Size повинен містити команди (500x500, 700x400 та 1024x720), що змінюють розмір графічного вікна. Пункт End повинен містити команди (закрити та скасувати).

27. ДІАЛОГОВІ ВІКНА В tkinter

Діалогові вікна, як елементи графічного інтерфейсу, призначені для виведення повідомлень користувачеві, отримання від нього будь-якої інформації, а також управління.

Діалогові вікна дуже різноманітні. Тут буде розглянуто лише кілька.

Розглянемо, як запрограмувати за допомогою tkinter виклик діалогових вікон відкриття та збереження файлів та роботу з ними. При цьому потрібно імпортувати «підмодуль» tkinter – `tkinter.filedialog`, в якому описані класи для вікон даного типу.

```
from tkinter import *
from tkinter.filedialog import *

root = Tk()
op = askopenfilename()
sa = asksaveasfilename()

root.mainloop()
```

Тут створюються два об'єкти (`op` та `sa`): один викликає діалогове вікно "Відкрити", а інший "Зберегти як...". При виконанні скрипту вони один за одним виводяться на екран після появи головного вікна. Якщо не створити root, то воно все одно з'явиться на екрані, проте при спробі його закриття в кінці виникне помилка.

Давайте розмістимо багаторядкове текстове поле на головному вікні і надалі спробуємо туди завантажувати вміст невеликих текстових файлів. Оскільки вікно збереження файлу нам поки що не потрібно, то закоментуємо цей рядок коду або видалимо. В результаті має вийти приблизно так:

```
from tkinter import *
from tkinter.filedialog import *
root = Tk()
txt = Text (root, width = 40, height = 15, font = "12")
txt.pack()
op = askopenfilename()
root.mainloop()
```

Під час запуску скрипту з'являється вікно з текстовим полем і відразу діалогове вікно "Відкрити". Однак, якщо ми спробуємо відкрити текстовий файл, то в кращому випадку нічого не станеться. Як зв'язати вміст текстового файлу з текстовим полем через діалог "Відкрити"?

Якщо просто вставити вміст змінної op в текстове поле:

```
txt.insert(END,op)
```

Після запуску скрипту та спроби відкриття файлу в текстовому полі надається адреса файлу. Значить, вміст файлу треба прочитати якимось методом (функцією).

Метод input модуля `fileinput` може приймати як аргумент адресу файлу, читати його вміст, формуючи список рядків. Далі за допомогою циклу for можна витягувати рядки послідовно та поміщати їх, наприклад, у текстове поле.

```
.....
import fileinput
.....
for i in fileinput.input(op):
    txt.insert(END,i)
.....
```

Зверніть увагу на те, як відбувається звернення до функції input модуля `fileinput` та його імпорту. Справа в тому, що в Python вже вбудована своя функція `input` (її призначення абсолютно інше) і, щоб уникнути "конфлікту", потрібно чітко вказати, яку саме функцію ми маємо на увазі. Тому варіант імпорту `from fileinput` та `import input` тут не підходить.

Вікно "Відкрити" запускається одразу при виконанні скрипту. Насправді так не має бути. Необхідно пов'язати запуск вікна з якоюсь подією. Нехай це буде клацання на пункті меню.

```
fromtkinter import *
from tkinter.filedialog import *
import fileinput
def_open():
   op = askopenfilename()
   for l in fileinput.input(op):
      txt.insert(END,l)
root = Tk()
m = Menu(root)
root.config(menu=m)
fm = Menu(m)
m.add_cascade(label="File",menu=fm)
fm.add_command(label="Open...",command=_open)
txt = Text (root, width = 40, height = 15, font = "12")
txt.pack()
root.mainloop()
```

Тепер спробуємо зберегти текст, набраний у текстовому полі. Додамо в код пункт меню та наступну функцію:

```
def_save():
   sa = asksaveasfilename()
   letter = txt.get(1.0,END)
   f = open(sa, "w")
   f.write(letter)
   f.close()
```

У змінній sa зберігається адреса файлу, куди буде записуватися. У змінній `letter` – текст, «отриманий» із текстового поля. Потім файл відкривається для запису, в нього записується вміст змінної `letter`, і файл закривається (про всяк випадок).

Ще одна група діалогових вікон описана у модулі `tkinter.messagebox`. Це досить прості діалогові вікна для виведення повідомлень, попереджень, отримання від користувача відповіді так чи ні тощо.

Доповнимо нашу програму пунктом Exit у підменю File і пунктом About program у підменю Help.

```
from tkinter.messagebox import *
def close_win():
   if askyesno("Exit", "Do you want to quit?"):
      root.destroy()
def about():
   showinfo("Editor", "This is text editor.\n
                                        (test version)")
fm.add_command(label="Exit",command=close_win)
hm = Menu(m)
m.add_cascade(label="Help",menu=hm)
hm.add_command(label="About",command=about)
...
```

У функції `about` відбувається виклик вікна `showinfo`, що дозволяє виводити повідомлення для користувача з кнопкою OK. Перший аргумент – це те, що виведеться в заголовку вікна, а другий – те, що буде у тілі повідомлення. У функції close_win викликається вікно `askyesno`, яке дозволяє отримати від користувача дві відповіді (`true` і `false`). У разі при позитивному відповіді спрацює гілка if головне вікно буде закрито. У разі натискання користувачем кнопки "No" вікно просто закриється (хоча можна було запрограмувати у гілці else будь-яку дію).

Питання для самоконтролю до теми 27

1. Що таке діалогові вікна Tkinter?
2. Який підмодуль Tkinter необхідно імпортувати для роботи з діалоговими вікнами Tkinter?
3. Які діалогові вікна Tkinter ви знаєте?
4. Яке призначення функції askopenfilename і asksaveasfilename?
5. Яке призначення функції askyesno і showinfo?

Завдання до теми 27

Для програми, розробленої у темі 26, додайте пункт меню FILE. Пункт має містити пункти OPEN SAVE CANCEL. Використовуйте функції askopenfilename() та asksaveasfilename() для вибору імені файлу. Ім'я вибраного файлу повідомте користувача, використовуючи функцію showinfo.

28. ГЕОМЕТРИЧНІ ПРИМІТИВИ CANVAS (ПОЛОТНО)

`Canvas` (полотно) – це досить складний об'єкт бібліотеки tkinter. Він дозволяє розташовувати на собі інші об'єкти. Це може бути як геометричні фігури, візерунки, вставлені зображення, і інші віджети (наприклад, мітки, кнопки, текстові поля). І це ще не все. Відображені на полотні об'єкти можна змінювати та переміщувати (за бажанням) у процесі виконання скрипту. Враховуючи все це, canvas знаходить широке застосування при створенні GUI-додатків з використанням tkinter (створення рисунків, оформлення інших віджетів, реалізація функцій графічних редакторів, програмована анімація та інше).

Буде розглянуто створення на полотні графічних примітивів (лінії, прямокутника, багатокутника, дуги (сектору), еліпса) та тексту.

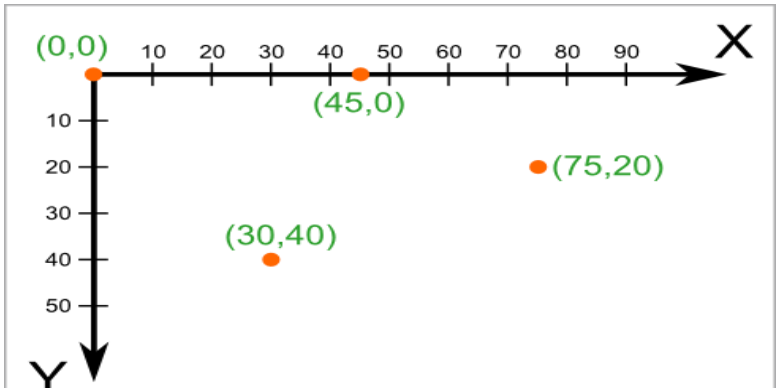
Для того, щоб створити об'єкт-полотно, необхідно викликати відповідний клас модуля tkinter і встановити деякі значення властивостей (опцій). Наприклад:

```
canv = Canvas(root,width=500,height=500,bg="lightblue",
                                                cursor = "pencil")
```

Далі за допомогою будь-якого менеджера з геометрії розмістити на головному вікні.

Перед тим, як створювати геометричні фігури на полотні, слід розібратися з координатами та одиницями вимірювання відстані. Нульова точка (0,0) для об'єкта `Canvas` розташована у верхньому лівому кутку. Одиниці виміру пікселі (точки екрану).

Для орієнтації в просторі об'єкта `Canvas` розгляньте малюнок нижче. У будь-якій точці перше число – це відстань від нульового значення по осі X, друге – по осі Y.



Щоб намалювати лінію на полотні слід до об'єкта (у нашому випадку canv) застосувати метод `create_line`.

```
canv.create_line(200,50,300,50,width=3,fill="blue")
canv.create_line(0,0,100,100,width=2,arrow=LAST)
```

Чотири числа – це пари координат початку та кінця лінії, тобто в прикладі перша лінія починається з точки (200,50), а закінчується в точці (300,50). Друга лінія починається в точці (0,0), закінчується – (100,100).

Властивість `fill` дозволяє задати колір лінії відмінний від чорного, а arrow – встановити стрілку (наприкінці, на початку або на обох кінцях лінії).

Метод `create_rectangle` створює прямокутник. Аналогічно лінії у дужках першими аргументами прописуються чотири числа. Перші дві координати позначають верхній лівий кут прямокутника, другі правий нижній. У прикладі нижче використовується дещо інший підхід. Він може бути корисним, якщо початкові координати об'єкта можуть змінюватися, яке розмір суворо регламентований.

```
x = 75
y = 110
canv.create_rectangle(x,y,x+80,y+50,fill="white",outline="blue")
```

Опція outline визначає колір межі прямокутника.

Щоб створити довільний багатокутник, потрібно задати пари координат кожної його точки.

```
canv.create_polygon([250,100],[200,150],[300,150], fill="yellow")
```

Квадратні дужки при заданні координат використовуються для зручності читання (їх можна не використовувати). Властивість `smooth` задає згладжування.

```
canv.create_polygon([250,100],[200,150],[300,150], fill="yellow")
canv.create_polygon([300,80],[400,80],
          [450,75],[450,200], [300,180],[330,160],
                              outline="white",smooth=1)
```

При створенні еліпса задаються координати гіпотетичного прямокутника, що описує цей еліпс.

```
canv.create_oval([20,200],[150,300],fill="gray50")
```

Більш складні розуміння фігури виходять під час використання методу create_arc. Залежно від значення опції `style` можна отримати сектор (за умовчанням), сегмент (`CHORD`) або дугу (`ARC`). Координати, як і раніше, задають прямокутник, в який вписано коло, з якого «вирізають» сектор, сегмент або дугу.

Від опцій start та extent залежить кут фігури.

```
canv.create_arc([160,230],[230,330],start=0,
                      extent=140, fill="lightgreen")
canv.create_arc([250,230],[320,330],start=0,
                   extent=140, style=CHORD,fill="green")

canv.create_arc([340,230],[410,330],start=0,
      extent=140,style=ARC,outline="darkgreen",width=2)
```

Метод об'єкта canvas `create_text` – це метод, що створює текстовий напис.

```
canv.create_text(20,330,
    text="Досліди з графічними примітивами\nна полотні",
        font="Verdana 12",anchor="w", justify=CENTER, fill="red")
```

Труднощі тут можуть виникнути з розумінням опції `anchor` (якір). За замовчуванням у заданій координаті розміщується центр текстового напису. Щоб змінити це та, наприклад, розмістити за вказаною координатою ліву межу

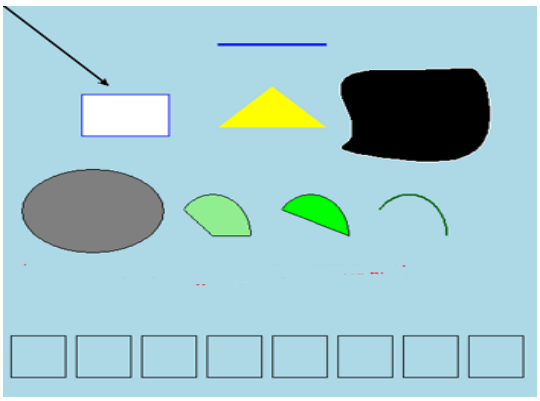
тексту, використовується якір зі значенням `w` (від англ. west – захід). Інші значення: `n, ne, e, se, s, sw, w, nw`.

Якщо букв, що задають бік прив'язки дві, то друга визначає вертикальну прив'язку (вгору чи вниз «піде» текст від координати). Властивість `justify` визначає лише вирівнювання тексту щодо себе самого.

Наприкінці слід зазначити, що часто потрібно «намалювати» на полотні будь-які елементи, що повторюються. Щоб не завантажувати код, використовують цикли. Наприклад, так:

```
x=10
while x < 450:
   canv.create_rectangle(x,400,x+50,450)
   x = x + 60
```

Якщо написати код наведений у даному тексте (попередньо здійснивши імпорт модуля tkinter і створення головного вікна, а також не забувши розташувати на вікні полотно, і в кінці "зробити" `mainloop`), то при його виконанні побачите таку картину:



Canvas (полотно) – методи, ідентифікатори та теги

Раніше були розглянуті методи об'єкта canvas, що формують на ньому геометричні примітиви та текст. Однак це лише частина методів полотна. В іншу умовну групу можна виділити методи, що змінюють властивості існуючих об'єктів полотна (наприклад, геометричних фігур). І тут постає питання: як звертатися до вже створених фігур? Адже якщо під час створення було прописано щось на кшталт `canvas.create_oval(30,10,130,80)` і таких овалів, квадратів та ін. на полотні дуже багато, то як до них звертатися?

Для вирішення цієї проблеми в tkinter для об'єктів полотна можна використовувати ідентифікатори та теги, які потім передаються іншим методам. Будь-який об'єкт може бути як ідентифікатор, так і тег. Використання ідентифікаторів та тегів трохи відрізняється.

Розглянемо кілька методів зміни вже існуючих об'єктів з використанням ідентифікаторів. Для початку створимо полотно та три об'єкти на ньому. При створенні об'єкти "повертають" свої ідентифікатори, які можна пов'язати зі змінними (`oval`, `rect` та `trial` у прикладі нижче) і потім використовувати їх для звернення до конкретного об'єкта.

```
c = Canvas(width=460,height=460,bg='grey80')
c.pack()
oval = c.create_oval(30,10,130,80)
rect = c.create_rectangle(180,10,280,80)
trian = c.create_polygon(330,80,380,10,430,80, fill='grey80',
                                                outline="black")
```

Якщо ви виконаєте цей скрипт, побачите на полотні три фігури: овал, прямокутник і трикутник.

Далі можна використовувати методи-модифікатори вказуючи в якості першого аргументу ідентифікатор об'єкта. Метод move переміщає об'єкт на осі X і Y на відстань зазначене як другий і третій аргументів. Слід розуміти, що це координати, а зміщення, т. е. у прикладі нижче прямокутник опуститься вниз на 150 пікселів. Метод itemconfig змінює зазначені властивості об'єктів, coords змінює координати (їм можна змінювати і розмір об'єкта).

```
c.move(rect,0,150)
c.itemconfig(trian,outline="red",width=3)
c.coords(oval,300,200,450,450)
```

Якщо запустити скрипт, що містить дві наведені частини коду (один за одним), то ми відразу побачимо картину, що вже змінилася, на полотні: прямокутник опуститься, трикутник придбає червоний контур, а еліпс зміститься і сильно збільшиться в розмірах. Зазвичай у програмах зміни повинні наступати при якомусь зовнішньому впливі. Нехай по клацанню лівою кнопкою миші прямокутник пересувається на два пікселі вниз (він буде робити це при кожному клацанні мишею):

```
def mooove(event):
   c.move(rect,0,2)
...
c.bind('<Button-1>',mooove)
```

Тепер розглянемо, як працюють теги. На відміну від ідентифікаторів, які є унікальними для кожного об'єкта, той самий тег може присвоюватися різним об'єктам. Подальше звернення до такого тегу дозволить змінити всі об'єкти, де він був вказаний. У прикладі нижче еліпс та лінія містять один і той же тег, а функція color змінює колір усіх об'єктів з тегом group1. Зверніть увагу, що на відміну від імені ідентифікатора (змінна), ім'я тегу полягає в лапки (рядкове значення).

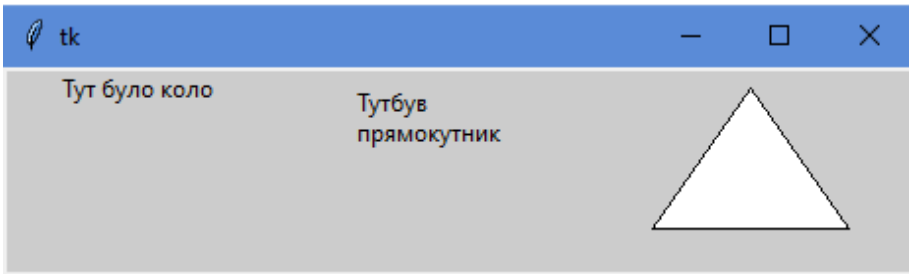
```
oval = c.create_oval(30,10,130,80,tag="group1")
c.create_line(10,100,450,100,tag="group1")
…
def color(event):
   c.itemconfig('group1',fill="red",width=3)
…
c.bind('<Button-3>',color)
```

Ще один метод, який варто розглянути, це `delete`, який видаляє об'єкт за вказаним ідентифікатором або тегом. У tinter є зарезервовані теги: наприклад, `all` позначає всі об'єкти полотна. Так, у прикладі нижче функція clean просто очищає полотно.

```
def clean(event):
   c.delete('all')
...
c.bind('<Button-2>',clean)
```

Метод `tag_bind` дозволяє прив'язати подію (наприклад, клацання кнопкою миші) до певного об'єкта. Таким чином, можна реалізувати звернення до різних областей полотна за допомогою однієї і тієї ж події. Приклад нижче це наочно ілюструє: зміни на полотні залежать від того, де зроблено клацання мишею.

```
from tkinter import *

c = Canvas(width=460,height=100,bg='grey80')
c.pack()

oval = c.create_oval(30,10,130,80,fill="orange")
c.create_rectangle(180,10,280,80,tag="rect", fill="lightgreen")

trian = c.create_polygon(330,80,380,10,430,80,
fill='white',outline="black")


def oval_func(event):
   c.delete(oval)
   c.create_text(30,10, text="Тут було коло",anchor="w")
def rect_func(event):
   c.delete("rect")
   c.create_text(180,10,
                 text="Тут був\nпрямокутник",anchor="nw")
def triangle(event):
   c.create_polygon(350,70,380,20,410,70,
                        fill='yellow',outline="black")

c.tag_bind(oval,'<Button-1>',oval_func)
c.tag_bind("rect",'<Button-1>',rect_func)
c.tag_bind(trian,'<Button-1>',triangle)

mainloop()
```



Особливості роботи із віджетом Text модуля Tkinter.

Графічний елемент `Text` надає великі можливості для роботи з текстовою інформацією. Крім різноманітних операцій з текстом та його форматуванням в екземпляр об'єкта `Text` можна вставляти інші віджети (слід зазначити, що така ж можливість існує і для `Canvas`). У цьому розглядаються лише деякі можливості віджету `Text` на прикладі створення вікна з текстовим полем, що містить форматований текст, кнопку та можливість додавання екземплярів полотна.

1. Для початку створимо текстове поле, встановивши при цьому деякі з його властивостей:

```
#Текстове поле та його початкові налаштування
tx = Text(font=('times',12), width =50,height =15,wrap=WORD)
tx.pack(expand=YES,fill=BOTH)
```

2. Тепер допустимо нам потрібно додати якийсь текст. Зробити це можна за допомогою методу `insert`, передавши йому два обов'язкові аргументи: місце, куди вставити, та об'єкт, який слід вставити. Об'єктом може бути рядок, змінний, що посилається на рядок або на будь-який інший об'єкт. Місце вставки може вказуватись декількома способами. Один із них – це індекси. Вони записуються як 'xy', де x – це рядок, а y – стовпець. У цьому нумерація рядків починається з одиниці, а стовпців з нуля. Наприклад, перший символ у першому рядку має індекс '1.0', а десятий символ у п'ятому рядку - '5.9'.

```
tx.insert(1.0,'Дзен Пітона\n\
 Якщо інтерпретатору Пітона дати команду\n\
 import this ("імпортувати це"),\n\
 то виведеться так званий \
  "Дзен Пітона".\n Деякі вирази:\n\
* Якщо реалізацію складно пояснити - це погана ідея.\n \
  Помилки ніколи не повинні замовчуватися.\n \
* Явне краще неявного.')
```

Комбінація символів `'\n'` створює новий рядок (тобто при інтерпретації наступний текст розпочнеться з нового рядка). Одиничний символ '\' ніяк не впливає на відображення тексту під час виконання коду, його слід вставляти при перенесенні тексту під час написання скрипту.

Якщо вміст текстового поля немає взагалі, то єдиний доступний індекс – '1.0'. У заповненому текстовому полі можна вставляти в будь-яке місце (де є вміст).

Якщо виконати скрипт, що містить тільки цей код (+ імпорт модуля tkinter, + створення головного вікна, + `mainloop`() наприкінці), ми побачимо текстове поле з вісьмома рядками тексту. Текст не оформлено.

3. Відформатуємо різні області тексту по-різному. Для цього спочатку задамо теги для потрібних нам областей, а потім для кожного тегу встановимо налаштування шрифту та інше.

```
#установка тегів для областей тексту
tx.tag_add('title','1.0','1.end')
tx.tag_add('special','6.0','8.end')
tx.tag_add('special','3.0','3.11')

#конфігурування тегів
tx.tag_config('title',foreground='red',
font=('times',14,'underline'),justify=CENTER)
tx.tag_config('special',background='grey85',
font=('Dejavu',10,'bold'))
```

Додавання тегу здійснюється за допомогою методу `tag_add`. Перший атрибут – ім'я тегу (довільне), далі за допомогою індексів вказується до якої області текстового поля він прикріплюється (початковий символ і кінцевий). Варіант запису як `'1.end'` говорить про те, що потрібно взяти текст до кінця вказаного рядка. Різні області тексту можуть бути позначені однаковим тегом.

Метод `tag_config` застосовує ті чи інші властивості до тегу, вказаного як перший аргумент.

4. У багаторядкове текстове поле можна додавати не лише текст, а й інші об'єкти. Наприклад, вставимо в поле кнопку (та й функцію заразом).

```
def erase():

   tx.delete('1.0',END)
...
#додавання кнопки
bt = Button(tx,text='Стерти',command=erase)
tx.window_create(END,window=bt)
```

Кнопка – це віджет. Віджети додаються в текстове поле за допомогою методу `window_create`, де як перша опція вказується місце додавання, а другий (`window`) – як значення присвоюється змінна, пов'язана з об'єктом.

При натисканні ЛКМ (лівої кнопки миші) по кнопці буде викликатися функція erase, в якій за допомогою методу delete видаляється весь вміст поля (від 1.0 до END).

5. А ось цікавіший приклад додавання віджету в поле Text:

```
def smiley(event):
  cv = Canvas (height = 30, width = 30)
  cv.create_oval(1,1,29,29,fill="yellow")
  cv.create_oval(9,10,12,12)
  cv.create_oval(19,10,22,12)
  cv.create_polygon(9,20,15,24,22,20)
  tx.window_create(CURRENT,window=cv)
…
#ЛКМ -> смайлик
```

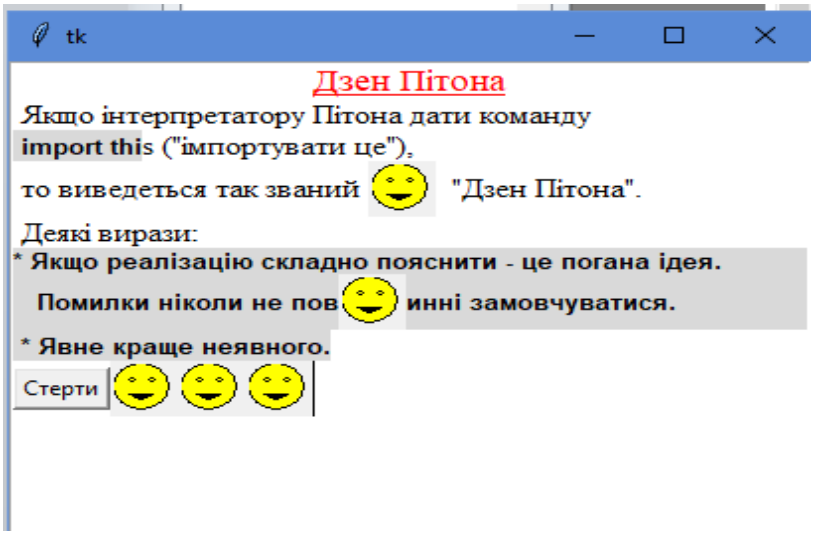
```
tx.bind('<Button-1>',smiley)
```

Тут при натисканні ЛКМ у будь-якому місці текстового поля буде викликатися функція smiley. У тілі цієї функції створюється об'єкт полотна, який наприкінці з допомогою методу window_create додається об'єкт tx. Місце вставки вказано як CURRENT, тобто "поточне" – це там, де було зроблено клацання мишею.

Нижче наведено текст описаного вище скрипту:

```
from tkinter import *

#Текстове поле та його початкові налаштування
tx = Text(font=('times',12), width =50,height=15,wrap=WORD)
tx.pack(expand=YES,fill=BOTH)

tx.insert(1.0, 'Дзен Пітона\n\
 Якщо інтерпретатору Пітона дати команду\n\
 import this ("імпортувати це"),\n\
 то виведеться так званий \
  "Дзен Пітона".\n Деякі вирази:\n\
* Якщо реалізацію складно пояснити - це погана ідея.\n \
  Помилки ніколи не повинні замовчуватися.\n \
* Явне краще неявного.\n')
#установка тегів для областей тексту
tx.tag_add('title','1.0','1.end')
tx.tag_add('special','6.0','8.end')
tx.tag_add('special','3.0','3.11')

#конфігурування тегів
tx.tag_config('title',foreground='red',
font=('times',14,'underline'),justify=CENTER)
tx.tag_config('special',background='grey85', \
font=('Dejavu',10,'bold'))


def erase():
   tx.delete('1.0',END)
...

#додавання кнопки
bt = Button(tx,text='Стерти',command=erase)

tx.window_create(END,window=bt)

def smiley(event):
  cv = Canvas (height = 30, width = 30)
  cv.create_oval(1,1,29,29,fill="yellow")
  cv.create_oval(9,10,12,12)
  cv.create_oval(19,10,22,12)
  cv.create_polygon(9,20,15,24,22,20)
  tx.window_create(CURRENT,window=cv)
```

```
#ЛКМ -> смайлик
tx.bind('<Button-1>',smiley)
```



Питання для самоконтролю до теми 28

1. Яке призначення об'єкту `Canvas` (полотно) бібліотеки tkinter?
2. Як створити об'єкт `Canvas` (полотно) бібліотеки tkinter?
3. Яка система координат використовується в об'єкті `Canvas` (полотно) бібліотеки Tkinter?
4. Які графічні примітиви можна розміщувати на об'єкті `Canvas` (полотно) бібліотеки Tkinter?
5. Які методи, ідентифікатори та теги існуючих об'єктів полотна ви знаєте?

Завдання до теми 28

Розробити програму, яка малює на формі (використовуйте `Canvas`) задані користувачем фігури. Фігури вибираються з меню користувача. Після вибору фігури визначте її розміри та місце розташування на формі за допомогою додаткового діалогового вікна.

29. ДЕКІЛЬКА ПРИКЛАДІВ

Гра «життя»

Гра "Життя" Джона Конвея – це клітинний автомат, створений математиком Джоном Конвеєм в 1970 році. Він моделює еволюцію клітин на нескінченному двовимірному полі, де кожна клітина може бути живою або мертвою. Стан поля змінюється крок за кроком за фіксованими правилами, які залежать від числа живих сусідів клітини.

Правила:

Виживання: Жива клітина з двома або трьома живими сусідами залишається живою.

Загибель: Жива клітина з менш ніж 2 (самотність) або більш ніж 3 сусідами (перенаселення) вмирає.

Народження: Мертва клітина з рівно 3 живими сусідами оживає.

Гра не має мети чи переможця — це математична модель, що показує, як із простих правил може виникати складна і непередбачувана поведінка (наприклад, стабільні фігури, осцилятори чи «кораблі», що рухаються полем)

```
from tkinter import Tk, Canvas, Button, Frame, BOTH, \
NORMAL, HIDDEN
# функція отримує координати миші
# в момент натискання або пересування із затиснутою кнопкою
def draw_a(e):
     ii = (e.y-3) / cell_size
     jj = (e.x-3) / cell_size
     ii = int (ii); jj = int (jj)
     print("e=",e, 'ii=',ii, 'jj=',jj)
     canvas.itemconfig(cell_matrix[addr(ii, jj)], \
                                        state=NORMAL, tags='vis')

# ця функція перетворює двовимірну координату в простий
# адресу одномірного масиву

def addr(ii,jj):
     if(ii < 0 or jj < 0 or ii >= field_height or jj >= \
                                                field_width):
          return int( len(cell_matrix)-1 )
     else:
          return int( ii * (win_width / cell_size) + jj )


# функція оновлення «картинки»
def refresh():
     for i in range(field_height):
          for j in range(field_width):
               k = 0
            ## вважаємо число сусідів
               for i_shift in range(-1, 2):
```

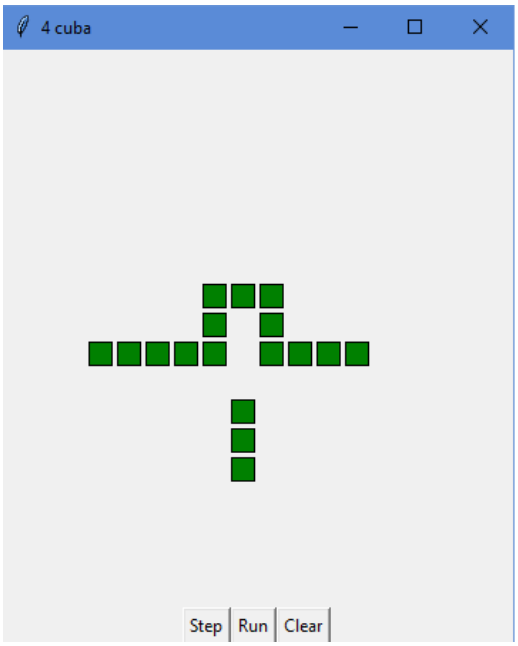
```
                    for j_shift in range(-1, 2):
                         if (canvas.gettags(
                                cell_matrix[addr(i + i_shift,
                                j + j_shift)])[0] == 'vis' and
                                 (i_shift != 0 or j_shift != 0)):
                              k += 1
               current_tag = \
                    canvas.gettags(cell_matrix[addr(i, j)])[0]
                # в залежності від їх числа
                #встановлюємо стан клітини
                #тут можна експериментувати
                # з самими "правилами" гри
               if(k == 3):
                    canvas.itemconfig(cell_matrix[addr(i, j)],
                                 tags=(current_tag, 'to_vis'))
               if(k <= 1 or k >= 4):
                    canvas.itemconfig(cell_matrix[addr(i, j)],
                                  tags=(current_tag, 'to_hid'))
               if(k == 2 and
                           canvas.gettags(cell_matrix[
                                   addr(i, j)])[0] == 'vis'):
                   canvas.itemconfig(cell_matrix[addr(i, j)],
                                  tags=(current_tag, 'to_vis'))
# сам крок: оновлюємо стан і малюємо
def step():
     refresh()
     repaint()

def clear():
     for i in range(field_height):
          for j in range(field_width):
               canvas.itemconfig(cell_matrix[addr(i, j)], \
                              state=HIDDEN, tags=('hid','0'))


# перемальовуємо поле по буферу
# згідно з другим тегом елемента
def repaint():
     for i in range(field_height):
          for j in range(field_width):
               if (canvas.gettags(cell_matrix[addr(i, j)])[1] == \
                                                   'to_hid'):
                    canvas.itemconfig(cell_matrix[addr(i, j)],
                                 state=HIDDEN, tags=('hid','0'))
               if (canvas.gettags(cell_matrix[addr(i, j)])[1] == \
                                                   'to_vis'):
                    canvas.itemconfig(cell_matrix[addr(i, j)], \
                                 state=NORMAL, tags=('vis','0'))

# «створення та підтримка виконання»
# автоматичного режиму

def loop():
```

```
    ##label['text'] = str(n)
    step()
    if (flag_run):
            root.after(500, loop) # call loop() in 0.5 seconds

# «Виконання» автоматичного режиму
def run():
        global flag_run, btn3
        #print('flag_run=',flag_run)
        if not flag_run:
                btn3.config(text="STOP")
                flag_run=True
                #print('stop')
                loop()
        else:
                btn3.config(text="RUN")
                flag_run=False
                #print('run')

flag_run=False # працює «автоматично» або по кроках

# створюємо саме вікно
root = Tk()
root.title('4 cuba')
# це ширина та висота вікна
win_width = 360 # 350
win_height = 380 # 370
# «будуємо» конфігураційний рядок – розмір вікна
config_string = "{0}x{1}".format(win_width, win_height + 32)
# Задаємо колір клітини
fill_color = "green"
#методом geometry() задаємо розміри, можна написати і
#просто рядок виду '360x380'
root.geometry(config_string)
#це ширина самої клітини, з урахуванням «просвіту»
cell_size = 20
# створюється об'єкт типу Canvas, на якому буде
#відбуватися безпосередньо малювання,

# він робиться дочірнім по відношенню до
# самого вікна root
canvas = Canvas (root, height = win_height)
# пакуємо його, аналог show() в інших системах
canvas.pack(fill=BOTH)
# Визначаємо розміри поля в клітинах
field_height = int( win_height / cell_size )
field_width = int( win_width / cell_size )
# створюємо масив для клітин, він одномірний,
# але використовуючи допоміжну функцію addr –
# Перетворюватимемо індекси 2-х мірний до 1-мірного
cell_matrix = []
# тут у циклі створюємо екземпляри клітин
# і робимо їх прихованими
```

```
for i in range(field_height):
     for j in range(field_width):
      # тут створюємо екземпляр клітини
      # і робимо його прихованими
          square = canvas.create_rectangle(2 + cell_size*j,
                  2 + cell_size*i,
                  cell_size + cell_size*j - 2,
                  cell_size + cell_size*i - 2, fill=fill_color)

          canvas.itemconfig(square, state=HIDDEN, tags=('hid','0'))

# «додаємо» до масиву
          cell_matrix.append(square)
# створимо фіктивний елемент,
# він як би «повсюди» поза полем

fict_square   =   canvas.create_rectangle(0,0,0,0,   state=HIDDEN,
tags=('hid','0'))


cell_matrix.append(fict_square)
# задаємо «клітини» які знаходяться на полі
# «за замовчуванням»

canvas.itemconfig(cell_matrix[addr(8, 8)], state=NORMAL, \
                                                  tags='vis')
canvas.itemconfig(cell_matrix[addr(10, 9)], state=NORMAL,\
                                                  tags='vis')
canvas.itemconfig(cell_matrix[addr(9, 9)], state=NORMAL, \
                                                  tags='vis')
canvas.itemconfig(cell_matrix[addr(9, 8)], state=NORMAL,\
                                                  tags='vis')
canvas.itemconfig(cell_matrix[addr(9, 7)], state=NORMAL,\
                                                  tags='vis')
canvas.itemconfig(cell_matrix[addr(10, 7)], state=NORMAL,\
                                                  tags='vis')



# створюємо кадр для зберігання кнопок і
# аналогічним чином, як з Canvas,

# встановлюємо кнопки дочірні кадри

frame = Frame(root)
# виконати «крок»
btn1 = Button (frame, text = 'Step', command = step) # Eval
# очистити поле
btn2 = Button (frame, text = 'Clear', command = clear)
# Зміна режимів «автоматично» - «по кроках»
btn3 = Button (frame, text = 'Run', command = run)
# пакуємо кнопки
```

```
btn1.pack(side='left')
btn2.pack(side='right')
btn3.pack(side='right')
# пакуємо кадр
frame.pack(side='bottom')

# прив'язуємо події кліка і руху миші над
# canvas до функції draw_a

canvas.bind('<B1-Motion>', draw_a)
canvas.bind('<ButtonPress>', draw_a)
# через 500 мкс передати управління функції loop
root.after(500, loop)
# стандартний цикл, що організовує події і
# загальну роботу віконного додатка

root.mainloop()
```

Вікно виведення програми



Програма для малювання

Розробимо дуже просту програму для малювання з використанням tkinter

Клас `Paint`

Створюємо клас і називаємо його "`Paint`".

Лістинг:

```
 from tkinter import *
from tkinter.messagebox import *

# клас Paint
class Paint(Frame):
    def __init__(self, parent):
        Frame.__init__(self, parent)
        self.parent = parent
```

```
# вихід із програми
def close_win():
  if askyesno("Вихід", "Ви впевнені?"):
    root.destroy()

# виведення довідки
def about():
  showinfo("Demo Paint", "Проста програма для малювання")

# функція для створення головного вікна
def main():
    global root
    root = Tk()
    root.geometry("800x600+300+300")
    app = Paint(root)
    m = Menu(root)
    root.config(menu=m)

    fm = Menu(m)
    m.add_cascade(label="Файл", menu=fm)
    fm.add_command(label="Вихід", command=close_win)

    hm = Menu(m)
    m.add_cascade(label="Довідка", menu=hm)
    hm.add_command(label="Про програму", command=about)
    root.mainloop()

if __name__ == "__main__":
    main()
```

Цей код створює каркас для майбутньої програми. Далі ми створюємо для нашого класу функцію `setUI`, в якій задаватимемо розташування всіх кнопок, міток та самого полотна – поля для малювання.

Лістинг:

```
def setUI(self):
  # Встановлюємо назву вікна
  self.parent.title("Demo Paint")
  # Розміщуємо активні елементи на батьківському вікні
  self.pack(fill=BOTH, expand=1)

  self.columnconfigure(6, weight=1)
  self.rowconfigure(2, weight=1)

  # Створюємо полотно з білим фоном
  self.canv = Canvas(self, bg="white")

  # Приклепуємо канвас методом grid.
  # Він буде знаходиться в 3-му ряду, першій колонці,
  # і буде займати 7 колонок, задаємо відступи
  # X і Y в 5 пікселів, і
```

```
# Примушуємо розтягуватися при розтягуванні
# всього вікна

self.canv.grid(row=2, column=0, columnspan=7, padx=5,\
                        pady=5, sticky=E + W + S + N)

# створюємо мітку для кнопок зміни кольору пензля
color_lab = Label(self, text="Колір: ")

# Встановлюємо створену мітку в перший рядок
# і першу колонку,
# задаємо горизонтальний відступ у 6 пікселів
color_lab.grid(row=0, column=0, padx=6)

# Створення кнопки: встановлення тексту кнопки,\
# завдання ширини кнопки (10 символів)
red_btn = Button(self, text="Червоний", width=10)

# встановлюємо кнопку в перший ряд, друга колонка
red_btn.grid(row=0, column=1)

# за аналогією створюємо інші кнопки
green_btn = Button(self, text="Зелений", width=10)
green_btn.grid(row=0, column=2)

blue_btn = Button(self, text="Синій", width=10)
blue_btn.grid(row=0, column=3)

black_btn = Button(self, text="чорний", width=10)
black_btn.grid(row=0, column=4)

white_btn = Button(self, text="Білий", width=10)
white_btn.grid(row=0, column=5)

# Створюємо мітку для кнопок зміни розміру пензля
size_lab = Label(self, text="Розмір кисті:")
size_lab.grid(row=1, column=0, padx=5)
one_btn = Button(self, text="2x", width=10)
one_btn.grid(row=1, column=1)

two_btn = Button(self, text="5x", width=10)
two_btn.grid(row=1, column=2)

five_btn = Button(self, text="7x", width=10)
five_btn.grid(row=1, column=3)

seven_btn = Button(self, text="10x", width=10)
seven_btn.grid(row=1, column=4)

ten_btn = Button(self, text="20x", width=10)
ten_btn.grid(row=1, column=5)

twenty_btn = Button(self, text="50x", width=10)
```

```
  twenty_btn.grid(row=1, column=6, sticky=W)
```

До методу __init__ потрібно додати виклик цього методу:

```
self.setUI()
```

Розробка методу `draw()`

Розробимо метод для малювання на полотні. У метод `__init__` потрібно додати параметри пензля за замовчуванням:

```
self.brush_size = 10
self.brush_color = "red"
self.color = "red"
```

Лістинг методу `draw()`:

```
def draw(self, event):
        self.canv.create_oval(event.x - self.brush_size,
                              event.y - self.brush_size,
                              event.x + self.brush_size,
                              event.y + self.brush_size,
                              fill=self.color,
                                     outline=self.color)
```

Малювання здійснюється шляхом створення кіл на полотні: користувач затискає ліву кнопку миші і під час руху мишею, по дорозі курсору будуть малюватись кола.

Залишилося лише прив'язати до полотна обробку щойно створеного методу. Код:

```
self.canv.bind("<B1-Motion>", self.draw)
```

Зміна кольору та розміру пензля

Розробимо метод зміни кольору пензля:

```
def set_color(self, new_color):
  self.color = new_color
```

Після цього до кожної кнопки верхнього ряду слід додати код обробки натискання цієї кнопки за наступним шаблоном:

```
red_btn = Button(self, text="Червоний", width=10, command=lambda:
self.set_color("red"))
red_btn.grid(row=0, column=1)
```

Код `command=lambda: self.set_color("red")` прив'язує функцію з потрібним аргументом до кнопки.

Використовуємо функцію `lambda`, тому що без `lambda` функція викликається відразу під час створення кнопки, а чи не лише її натисканні.

Розглянемо метод зміни розміру пензля:

```
def set_brush_size(self, new_size):
  self.brush_size = new_size
```

Модернізуємо код кожної кнопки нижнього ряду за наступним шаблоном:

```
one_btn = Button(self, text="2x", width=10, command=lambda:
self.set_brush_size(2))
```

Залишилось додати кнопку “Очистити” для нашої програми. Дана кнопка очищатиме полотно. Лістинг:

```
clear_btn = Button(self, text="Очистити", width=10,
command=lambda: self.canv.delete("all"))
clear_btn.grid(row=0, column=6, sticky=W)
```

Повний лістинг програми

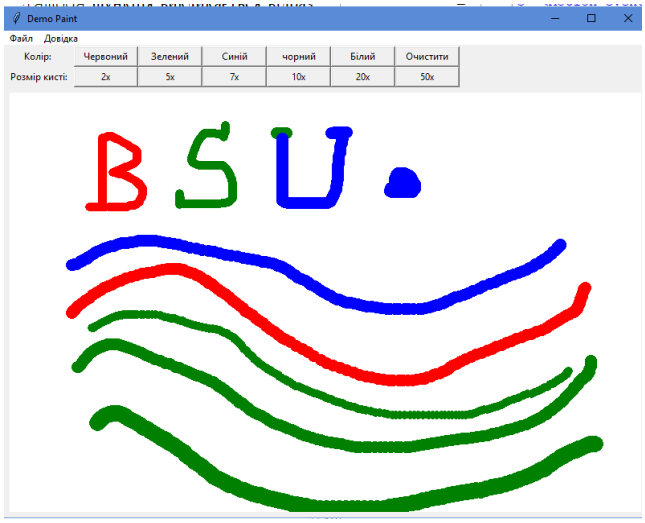
```
from tkinter import *
from tkinter.messagebox import *
#на основі коду з https://it-black.ru
# клас Paint
class Paint(Frame):
    def __init__(self, parent):
        Frame.__init__(self, parent)
        self.parent = parent
        # Параметри пензля за замовчуванням
        self.brush_size = 10
        self.brush_color = "red"
        self.color = "red"
        # Встановлюємо компоненти UI
        self.setUI()

        # Метод малювання на полотні

    def draw(self, event):
        self.canv.create_oval(event.x - self.brush_size,
                              event.y - self.brush_size,
                              event.x + self.brush_size,
                              event.y + self.brush_size,
                              fill=self.color, outline=self.color)

        # Зміна кольору пензля
    def set_color(self, new_color):
        self.color = new_color

        # Зміна розміру пензля
    def set_brush_size(self, new_size):
        self.brush_size = new_size

    def setUI(self):
        # Встановлюємо назву вікна
        self.parent.title("Demo Paint")
        # Розміщуємо активні елементи на батьківському вікні
        self.pack(fill=BOTH, expand=1)

        self.columnconfigure(6, weight=1)
        self.rowconfigure(2, weight=1)

        # Створюємо полотно з білим тлом
        self.canv = Canvas(self, bg="white")
```

```
        # Приклепуємо канвас методом grid.
        #Він буде знаходиться в 3-му ряду, першій колонці,
        # і буде займати 7 колонок, задаємо відступи X і Y
        #                                    в 5 пікселів, і
        # Примушуємо розтягуватися при розтягуванні всього вікна

        self.canv.grid(row=2, column=0, columnspan=7, padx=5, \
                                    pady=5, sticky=E + W + S + N)

        # задаємо реакцію полотна на натискання лівої кнопки миші
        self.canv.bind("<B1-Motion>", self.draw)

        # створюємо мітку для кнопок зміни кольору пензля
        color_lab = Label(self, text="Колір: ")

        # Встановлюємо створену мітку в перший рядок
        # і першу колонку,
        # задаємо горизонтальний відступ у 6 пікселів
        color_lab.grid(row=0, column=0, padx=6)

        # Створення кнопки: встановлення тексту кнопки,
        # завдання ширини кнопки (10 символів)
        red_btn = Button(self, text="Червоний", width=10, \
                         command=lambda: self.set_color("red"))

        # встановлюємо кнопку в перший ряд, друга колонка
        red_btn.grid(row=0, column=1)

        # за аналогією створюємо інші кнопки
        green_btn = Button(self, text="Зелений", width=10,
command=lambda: self.set_color("green"))
        green_btn.grid(row=0, column=2)

        blue_btn = Button(self, text="Синій", width=10,
command=lambda: self.set_color("blue"))
        blue_btn.grid(row=0, column=3)

        black_btn = Button(self, text="чорний", width=10,
command=lambda: self.set_color("black"))
        black_btn.grid(row=0, column=4)

        white_btn = Button(self, text="Білий", width=10,
command=lambda: self.set_color("white"))
        white_btn.grid(row=0, column=5)

        # Створюємо мітку для кнопок зміни розміру пензля
        size_lab = Label(self, text="Розмір кисті:")
        size_lab.grid(row=1, column=0, padx=5)
        one_btn = Button(self, text="2x", width=10, \
                         command=lambda: self.set_brush_size(2))
        one_btn.grid(row=1, column=1)
```

```
        two_btn = Button(self, text="5x", width=10, \
                         command=lambda: self.set_brush_size(5))
        two_btn.grid(row=1, column=2)

        five_btn = Button(self, text="7x", width=10, \
                         command=lambda: self.set_brush_size(7))
        five_btn.grid(row=1, column=3)

        seven_btn = Button(self, text="10x", width=10, \
                           command=lambda: self.set_brush_size(10))
        seven_btn.grid(row=1, column=4)

        ten_btn = Button(self, text="20x", width=10, \
                           command=lambda: self.set_brush_size(20))
        ten_btn.grid(row=1, column=5)

        twenty_btn = Button(self, text="50x", width=10, \
                         command=lambda: self.set_brush_size(50))
        twenty_btn.grid(row=1, column=6, sticky=W)

        clear_btn = Button(self, text="Очистити", width=10, \
                           command=lambda: self.canv.delete("all"))
        clear_btn.grid(row=0, column=6, sticky=W)

# вихід із програми
def close_win():
  if askyesno("Вихід", "Ви впевнені?"):
    root.destroy()

# висновок довідки
def about():
  showinfo("Demo Paint", "Проста програма для малювання")

#функція для створення головного вікна

def main():
    global root
    root = Tk()
    root.geometry("800x600+300+300")
    app = Paint(root)
    m = Menu(root)
    root.config(menu=m)

    fm = Menu(m)
    m.add_cascade(label="Файл", menu=fm)
    fm.add_command(label="Вихід", command=close_win)

    hm = Menu(m)
    m.add_cascade(label="Довідка", menu=hm)
    hm.add_command(label="Про програму", command=about)
    root.mainloop()

if __name__ == "__main__":
```

```
main()
```

Буде виведено (і намальовано):



Для повного розуміння програми рекомендується читати коментарі до коду. Програма дуже проста і тому рекомендується самостійно доопрацювати або переробити код програми.

Питання для самоконтролю до теми 29

1. Який сценарій гри життя?
2. Як створюється та підтримується виконання кроків гри в автоматичному режимі?
3. Як збільшити розмір ігрового поля?
4. Як змінити колір клітини?
5. Що здійснює виклик методу root.after(500, loop)?
6. Додайте в код "малювальниці" можливість збереження малюнка.

Завдання до теми 29

Для програми малювання додайте можливість малювання стандартних фігур. Фігури вибираються з меню користувача. Після вибору фігури визначте її розміри та місце розташування на формі за допомогою додаткового діалогового вікна. Додайте до програми можливість збереження намальованого.

30. СИМВОЛЬНІ ОБЧИСЛЕННЯ МОВОЮ PYTHON.

SymPy є відкритою бібліотекою символьних обчислень мовою Python. SymPy повністю написана мовою Python і не вимагає сторонніх бібліотек.

Нижче, у пізнавальній меті, описані основні можливості пакету SymPy.

Детальніше дивіться http://www.sympy.org/en/index.html

Математична бібліотека Python `SymPy`

`SymPy` – це бібліотека Python для виконання символьних обчислень. Це система комп'ютерної алгебри, яка може виступати як окрема програма, так і як бібліотека для інших програм.

У `SymPy` є різні функції, які застосовуються у сфері символьних обчислень, математичного аналізу, алгебри, дискретної математики, квантової фізики тощо. SymPy може представляти результат у різних форматах: LaTeX, MathML і таке інше. Поширюється бібліотека ліцензії New BSD. Першими цю бібліотеку випустили розробники Ondřej Čertík та Aaron Meurer у 2007 році.

Ось де застосовується `SymPy`:

- Багаточлени
- Математичний аналіз
- Дискретна математика
- Матриці
- Геометрія
- Побудова графіків
- Фізика
- Статистика
- Комбінаторика

Установка `SymPy`

Для роботи `SymPy` потрібна одна важлива бібліотека під назвою mpmath. Вона використовується для речової та комплексної арифметики з числами з плаваючою точкою довільної точності. Однак pip встановить її автоматично під час завантаження самої SymPy:

```
python -m pip install sympy
```

Використання SymPy як калькулятор

SymPy підтримує три типи чисельних даних: `Float`, `Rational` та `Integer`.

`Rational` являє собою звичайний дріб, який задається за допомогою двох цілих чисел: чисельник і знаменник. Наприклад, `Rational(1, 2)` є дріб 1/2, `Rational(5, 2)` є дріб 5/2, тощо.

```
>>> from sympy import Rational
>>> a = Rational(1, 2)

>>> a
1/2

>>> a*2
1

>>> Rational(2)**50/Rational(10)**50
1/888178419700125232338905334472656 25
```

Важлива особливість Python-інтерпретатора – при розподілі двох "пітонівських" чисел типу int за допомогою оператора "/" виходить "пітонівський" тип `float`. Цей стандарт “true division” за замовчуванням включений і в sympy:

```
>>> 1/2
0.5
```

Зверніть увагу, що і в тому, і в іншому випадку ви маєте справу не з об'єктом Number з бібліотеки SymPy, який представляє число в SymPy, а з пітонівськими числами, які створюються самим інтерпретатором Python. Швидше за все, вам потрібно буде працювати з дробовими числами з бібліотеки SymPy, тому для того, щоб отримувати результат у вигляді об'єктів SymPy, переконайтеся, що ви використовуєте клас `Rational`. Комусь може здатися зручним позначати `Rational` як R:

```
import sympy
R = sympy.Rational
R(1, 2)
R(1)/2
```

У модулі Sympy є спеціальні константи, такі як `e` і `pi`, які поводяться як змінні (тобто вираз 1 + pi не перетворюється відразу в число, а так і залишиться 1 + pi):

```
>>> from sympy import pi, E
>>> pi**2
pi**2

>>> pi.evalf()
3.14159265358979

>>> (pi + E).evalf()
5.85987448204884
```

як видно, функція `evalf` переводить вихідний вираз у число з точкою, що плаває. Обчислення можна проводити з більшою точністю. Для цього потрібно передати як аргумент цієї функції необхідну кількість десяткових знаків:

```
>>> pi.evalf(100)
```

```
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482538
2
>>>
```

Для роботи з математичною нескінченністю використовується символ oo:

```
>>> from sympy import oo
>>> oo > 99999
True
>>> oo + 1
oo
```

Змінні

На відміну від інших систем комп'ютерної алгебри, потрібно явно декларувати символьні змінні:

```
>>> from sympy import symbols
>>> x = symbols('x')
>>> y = symbols('y')
```

У лівій частині цього виразу знаходиться змінна Python, яка пітонівським присвоєнням співвідноситься з об'єктом класу Symbol із SymPy.

```
>>> з sympy.abc import x, theta
```

Символьні змінні можуть також задаватися і за допомогою функцій symbols або var. Вони допускають вказівку діапазону. Їхня відмінність полягає в тому, що var додає створені змінні в поточний простір імен:

```
>>> from sympy import symbols, var
>>> a, b, c = symbols('a,b,c')
>>> d, e, f = symbols('d:f')
>>> var('g:h')
(g, h)
>>> var('g:2')
(g0, g1)
```

Примірники класу Symbol взаємодіють один з одним. Таким чином, за допомогою них конструюються алгебраїчні вирази:

```
>>> x + y + x - y
2*x

>>> (x + y)**2
(x + y)**2
#розкрити дужки
>>> ((x + y)**2).expand()
x**2 + 2*x*y + y**2
```

Змінні можуть бути замінені іншими змінними, числами або виразами за допомогою функції підстановки `subs(old, new)`:

```
>>> ((x + y)**2).subs(x, 1)
(y + 1)**2

>>> ((x + y)**2).subs(x, y)
4*y**2

>>> ((x + y)**2).subs(x, 1 - y)
1
```

Тепер, з цього моменту, для всіх наведених нижче прикладів ми будемо припускати, що запустили наступну команду з налаштування системи відображення результатів (і використовуємо процедуру pprint):

```
>>> from sympy import init_printing
>>> init_printing(use_unicode=False, wrap_line=False,
                                                  no_global=True)
```

Вона надасть якіснішого відображення виразів. Докладніше про систему відображення та друку наведено нижче в розділі Друк. Якщо ж шрифт встановлений з юнікодом, то можна використовувати опцію `use_unicode=True` для ще красивішого виведення.

Алгебра

Щоб розкласти вираз на найпростіші дроби, використовується функція `apart(expr, x)`:

```
#apart(expr, x):
from sympy import apart
from sympy.abc import x, y, z
from sympy import Integral, pprint
z= 1/( (x + 2)*(x + 1) )
pprint(z)
z=apart(1/( (x + 2)*(x + 1) ), x)
pprint(z)
z=apart((x + 1)/(x - 1), x)
pprint(z)
```

Висновок скрипту:

```
       1
---------------
(x + 1)*(x + 2)

    1       1
- ----- + -----
  x + 2   x + 1

      2
1 + -----
    x - 1
```

Щоб привести дріб до спільного знаменника, використовується функція `together(expr, x)`:

```
>>> from sympy import together
>>> from sympy.abc import x, y, z

>>> together(1/x + 1/y + 1/z)
x*y + x*z + y*z
---------------
     x*y*z

>>> together(apart((x + 1)/(x - 1), x), x)
```

```
x + 1
-----
x - 1

>>> together(apart(1/( (x + 2)*(x + 1) ), x), x)
       1
---------------
(x + 1)*(x + 2)
```

Обчислення

Межі

Для обчислення межі функції використовуйте функцію `limit(function, variable, point).`

Наприклад, щоб обчислити межу `f(x)` при `x -> 0`, потрібно ввести `limit(f, x, 0):`

```
>>> from sympy import limit, Symbol, sin, oo
>>> x = Symbol("x")
>>> limit(sin(x)/x, x, 0)
1
```

також можна обчислювати межі при x, що прагне нескінченності:

```
>>> limit(x, x, oo)
oo

>>> limit(1/x, x, oo)
0

>>> limit(x**x, x, 0)
1
```

Диференціювання

Продиференціювати будь-який вираз SymPy можна за допомогою `diff(func, var).`

Приклади:

```
>>> from sympy import diff, Symbol, sin, tan
>>> x = Symbol('x')
>>> diff(sin(x), x)
cos(x)
>>> diff(sin(2*x), x)
2*cos(2*x)

>>> diff(tan(x), x)
2
tan(x) + 1
```

Можна через межі перевірити правильність обчислень похідної:

```
>>> from sympy import limit
>>> from sympy.abc import delta
>>> limit((tan(x + delta) - tan(x))/delta, delta, 0)
```

```
   2
tan(x) + 1
```

Похідні вищих порядків можна обчислити, використовуючи додатковий параметр цієї функції `diff(func, var, n)`:

```
>>> diff(sin(2*x), x, 1)
2*cos(2*x)

>>> diff(sin(2*x), x, 2)
-4*sin(2*x)

>>> diff(sin(2*x), x, 3)
-8*cos(2*x)
```

Розкладання в ряд

Для розкладання в ряд використовуйте метод `series(var, point, order)`:

```
>>> from sympy import Symbol, cos
>>> x = Symbol('x')
>>> cos(x).series(x, 0, 10)

1 - x**2/2 + x**4/24 - x**6/720 + x**8/40320 + O(x**10)

>>> (1/cos(x)).series(x, 0, 10)

1 + x**2/2 + 5*x**4/24 + 61*x**6/720 + 277*x**8/8064 +
                                                  O(x**10)
```

Ще один простий приклад:

```
>>> from sympy import Integral, pprint

>>> y = Symbol("y")
>>> e = 1/(x + y)
>>> s = e.series(x, 0, 5)

>>> print(s)
1/y - x/y**2 + x**2/y**3 - x**3/y**4 + x**4/y**5 + O(x**5)


>>> pprint(s)

           2    3    4     ⎛ 5⎞
1   x    x    x    x      ⎜x ⎟
─ - ── + ── - ── + ── + O⎝  ⎠
y    2    3    4    5
    y    y    y    y
```

Суми

Обчислює значення суми f(від заданої змінної) на заданих межах

```
    summation(f, (i, a, b))-
```

Знаходження суми доданків f, і змінюється від a до b:

```
                                  b
                                ____
                                \   `
summation(f, (i, a, b)) =        )     f
                                /___,
                                i = a
```

Якщо він не може `compute sum`, його деталі відповідають формула підсумовування. Повторні суми можна обчислити, ввівши додаткові обмеження. Якщо сума не розраховується, то друкується відповідна формула:

```
>>> from sympy import summation, oo, symbols, log
>>> i, n, m = symbols('i n m', integer=True)

>>> summation(2*i - 1, (i, 1, n))
n**2
>>> summation(1/2**i, (i, 0, oo))
2

>>>z = summation(1/log(n)**n, (n, 2, oo))
Sum(log(n)**(-n), (n, 2, oo))
>>> pprint(z)
  ∞
 ___
 \
  \      -n
  /   log  (n)
 /
 ---
n = 2
>>> summation(i, (i, 0, n), (n, 0, m))
m**3/6 + m**2/2 + m/3
>>>pprint(z)
 3    2
m    m    m
── + ── + ─
6    2    3

>>> from sympy.abc import x
>>> from sympy import factorial
>>> summation(x**n/factorial(n), (n, 0, oo))
z=summation(x**n/factorial(n), (n, 0, oo))
z
exp(x)
pprint(z)
```

Інтегрування

SymPy підтримує обчислення певних та невизначених інтегралів за допомогою функції `integrate()`. Вона використовує розширений алгоритм Ріша-Нормана та деякі шаблони та евристики. Можна обчислювати інтеграли трансцендентних, простих та спеціальних функцій:

```
>>> from sympy import integrate, erf, exp, sin,
                                          log, oo, pi, sinh, symbols
>>> x, y = symbols('x,y')
```

Ви можете інтегрувати найпростіші функції:

```
>>> integrate(6*x**5, x)
  6
x
>>> integrate(sin(x), x)
-cos(x)
>>> integrate(log(x), x)
x * log (x) - x
>>> integrate(2*x + sinh(x), x)
  2
x + cosh(x)
```

Приклади інтегрування деяких спеціальних функцій:

```
>>> z=integrate(exp(-x**2)*erf(x), x)
>>> print(z)

sqrt(pi)*erf(x)**2/4
```

Можна також обчислити певний інтеграл:

```
>>> integrate(x**3, (x, -1, 1))
0
>>> integrate(sin(x), (x, 0, pi/2))
1
>>> integrate(cos(x), (x, -pi/2, pi/2))
2
```

Підтримуються і невласні інтеграли:

```
>>> integrate(exp(-x), (x, 0, oo))
1
>>> integrate(log(x), (x, 0, 1))
-1
```

Комплексні числа

Крім уявної одиниці `I`, що є уявним числом, символи теж можуть мати спеціальні атрибути (`real`, `positive`, `complex` тощо), які визначають поведінку цих символів при обчисленні символьних виразів:

```
>>> from sympy import Symbol, exp, I
>>> x = Symbol("x") # a plain x with no attributes
```

```
>>> exp(I*x).expand()
I*x
e
>>> exp(I*x).expand(complex=True)

-im(x) -im(x)
I * e * sin (re (x)) + e * cos (re (x))

>>> x = Symbol("x", real=True)
>>> exp(I*x).expand(complex=True)

I * sin (x) + cos (x)
```

Функції тригонометричні

```
>>> from sympy import asin, asinh, cos, sin, sinh, symbols, I
>>> x, y = symbols('x,y')

>>> sin(x + y).expand(trig=True)
sin(x)*cos(y) + sin(y)*cos(x)

>>> cos(x + y).expand(trig=True)
-sin(x)*sin(y) + cos(x)*cos(y)

>>> sin(I*x)
I*sinh(x)

>>> sinh(I*x)
I*sin(x)

>>> asinh(I)
I*pi
----
 2

>>> asinh(I*x)
I*asin(x)

>>> sin(x).series(x, 0, 10)

x - x**3/6 + x**5/120 - x**7/5040 + x**9/362880 + O(x**10)

>>> sinh(x).series(x, 0, 10)

x + x**3/6 + x**5/120 + x**7/5040 + x**9/362880 + O(x**10)

>>> asin(x).series(x, 0, 10)

x + x**3/6 + 3*x**5/40 + 5*x**7/112 + 35*x**9/1152 + O(x**10)
```

Факторіали та гамма-функції

```
>>> from sympy import factorial, gamma, Symbol
>>> x = Symbol("x")
>>> n = Symbol("n", integer=True)

>>> factorial(x)
x!

>>> factorial(n)
n!

>>> gamma(x + 1).series(x, 0, 3) # i.e. factorial(x)

1 - EulerGamma*x + x**2*(EulerGamma**2/2 + pi**2/12) + O(x**3)

                      /          2     2\
                    2 |EulerGamma    pi |    / 3\
1 - EulerGamma*x + x *|----------- + ---| + O\x /
                      \     2         12/
```

Дзета - функції

```
>>> from sympy import zeta
>>> zeta(4, x)
zeta(4, x)

>>> zeta(4, 1)

  4
pi
---
 90

>>> zeta(4, 2)

       4
     pi
-1 + ---
      90

>>> zeta(4, 3)

          4
  17   pi
- -- + ---
  16    90
```

Багаточлени

```
>>> from sympy import assoc_legendre, chebyshevt, legendre,
                                                          hermite
>>> chebyshevt(2, x)
```

```
   2
2*x  - 1

>>> chebyshevt(4, x)

   4      2
8*x  - 8*x  + 1

>>> legendre(2, x)
   2
3*x    1
---- - -
 2     2

>>> legendre(8, x)

      8          6          4         2
6435*x    3003*x    3465*x    315*x     35
------- - ------- + ------- - ------ + ---
  128         32         64        32    128

>>> assoc_legendre(2, 1, x)

        _________
       /     2
-3*x*\/  - x  + 1


>>> assoc_legendre(2, 2, x)

     2
- 3*x  + 3

>>> hermite(3, x)

   3
8*x  - 12*x
```

Диференціальні рівняння

У sympy:

```
>>> from sympy import Function, Symbol, dsolve
>>> f = Function('f')
>>> x = Symbol('x')
>>> f(x).diff(x, x) + f(x)
f(x) + Derivative(f(x), (x, 2))
>>> from sympy import Integral, pprint
>>> uuu=f(x).diff(x, x) + f(x)
>>> pprint(uuu)
```

```
          2
         d
f(x) + ---(f(x))
          2
        dx

>>> dsolve(f(x).diff(x, x) + f(x), f(x))
Eq(f(x), C1*sin(x) + C2*cos(x))
>>> ud=dsolve(f(x).diff(x, x) + f(x), f(x))
>>> ud
Eq(f(x), C1*sin(x) + C2*cos(x))
>>> pprint(ud)
f(x) = C1*sin(x) + C2*cos(x)
```

Алгебраїчні рівняння

```
>>> from sympy import solve, symbols
>>> x, y = symbols('x,y')
>>> solve(x**4 - 1, x)
[-1, 1, -I, I]

>>> solve([x + 5*y - 2, -3*x + 6*y - 15], [x, y])
{x: -3, y: 1}
```

Лінійна алгебра. Матриці

Матриці задаються за допомогою конструктора Matrix:

```
>>> from sympy import Matrix, Symbol
>>> Matrix([[1, 0], [0, 1]])
[1 0]
[]
[0 1]
```

У матрицях ви також можете використовувати символьні змінні:

```
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> A = Matrix([[1, x], [y, 1]])
>>> A
[1  x]
[    ]
[y  1]

>>> A**2
[x*y + 1    2*x  ]
[                ]
[  2*y    x*y + 1]
```

Зіставлення зі зразком

Щоб зіставити вирази зі зразками, використовуйте .match() разом із допоміжним класом Wild. Ця функція поверне словник із необхідними замінами, наприклад:

```
>>> from sympy import Symbol, Wild
```

```
>>> x = Symbol('x')
>>> p = Wild('p')
>>> (5*x**2).match(p*x**2)
{p: 5}

>>> q = Wild('q')
>>> (x**2).match(p*x**q)
{p: 1, q: 2}
```

Якщо ж зіставлення не вдалося, функція поверне "None":

```
>>> print((x + 1).match(p**x))
None
```

Також можна використовувати параметр exclude для виключення деяких значень із результату:

```
>>> p = Wild('p', exclude=[1, x])
>>> print((x + 1).match(x + p)) # 1 is excluded
None
>>> print((x + 1).match(p + 1)) # x is excluded
None
>>> print((x + 1).match(x + 2 + p)) # -1 is not excluded
{p_: -1}
```

Друк

Реалізовано кілька способів виведення виразів на екран.

Стандартний

Стандартний спосіб представлений функцією str(expression), яка працює наступним чином:

```
>>> from sympy import Integral
>>> from sympy.abc import x
>>> print(x**2)
x**2
>>> print(1/x)
1/x
>>> print(Integral(x**2, x))
Integral(x**2, x)
```

«Гарний друк»

Цей спосіб друку виразів заснований на ascii-графіці та реалізований через функцію pprint:

```
>>> from sympy import Integral, pprint
>>> from sympy.abc import x

 2
X

>>> pprint(1/x)

1
-
X
```

```
>>> pprint(Integral(x**2, x))
  /
 |
 |  2
 | x  dx
 |
/
```

Якщо у вас встановлено шрифт із юнікодом, він буде використовувати Pretty-print із юнікодом за замовчуванням. Це налаштування можна вимкнути за допомогою `use_unicode`:

```
>>> pprint(Integral(x**2, x), use_unicode=True)
⌠
⎮ 2
⎮ x dx
⌡
```

Для того, щоб зробити pprint за замовчуванням у стандартному інтерпретаторі, виконуємо таку процедуру:

```
>>> from sympy import *
>>>import sys
>>>sys.displayhook = pprint
```

Приклади:

```
>>> from sympy import init_printing, var, Integral
>>> init_printing(use_unicode=False, \
                      wrap_line=False, no_global=True)

>>> var("x")
X

>>> x**3/3

 3
x
--
3

>>> Integral(x**2, x) #doctest: +NORMALIZE_WHITESPACE
  /
 |
 |  2
 | x  dx
 |
/
```

Друк об'єктів Python

```
>>> from sympy.printing.python import python
>>> from sympy import Integral
>>> from sympy.abc import x
```

```
>>> print(python(x**2))
x = Symbol('x')
e = x**2
>>> print(python(1/x))
x = Symbol('x')
e = 1/x
>>> print(python(Integral(x**2, x)))
x = Symbol('x')
e = Integral(x**2, x)
```

Друк у форматі LaTeX

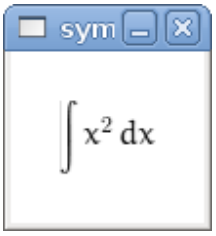
```
>>> from sympy import Integral, latex
>>> from sympy.abc import x
>>> latex(x**2)
x^{2}
>>> latex(x**2, mode='inline')
$x^{2}$
>>> latex(x**2, mode='equation')
\begin{equation}x^{2}\end{equation}
>>> latex(x**2, mode='equation*')
\begin{equation*}x^{2}\end{equation*}
>>> latex(1/x)
\frac{1}{x}
>>> latex(Integral(x**2, x))
\int x^{2}\, dx
```

MathML

```
>>> from sympy.printing.mathml import mathml
>>> from sympy import Integral, latex
>>> from sympy.abc import x
>>> print(mathml(x**2))
<apply><power/><ci>x</ci><cn>2</cn></apply>
>>> print(mathml(1/x))
<apply><power/><ci>x</ci><cn>-1</cn></apply>
```

Pyglet

```
>>> from sympy import Integral, preview
>>> from sympy.abc import x
>>> preview(Integral(x**2, x))
```



Примітки

Якщо встановлені додаткові пакети `Isympy` викликає pprint автоматично, тому Pretty-print буде включено в isympy за замовчуванням.

Також доступний модуль друку – `sympy.printing`. Через цей модуль доступні наступні функції друку:

`pretty(expr), pretty_print(expr), pprint(expr)` –

Повертає або виводить на екран, відповідно, "Гарне" подання виразу expr.

`latex(expr), print_latex(expr)` –

Повертає або виводить на екран, відповідно, LaTeX-подання expr

`mathml(expr), print_mathml(expr)` –

Повертає або виводить на екран, відповідно, MathML подання `expr`.

Приклади застосування пакету SymPy

SymPy – це пакет для символьних обчислень на пітоні, подібний до системи Mathematica. Він працює з виразами, що містять символи.

```
from sympy import *
init_printing()
```

Основними цеглинами, з яких будуються вирази, є символи. Символ має ім'я, яке використовується для друку виразів. Об'єкти класу Symbol потрібно створювати та присвоювати змінним пітонам, щоб їх можна було використовувати.

В принципі, ім'я символу та ім'я змінної, якою ми присвоюємо цей символ, – дві незалежні речі, і можна написати `abc=Symbol('xyz')`.

Але тоді при введенні програми будете використовувати `abc`, а під час друкування результатів SymPy буде використовувати `xyz`, що призведе до непотрібної плутанини. Тому краще, щоб ім'я символу збігалося з ім'ям змінної пітона, якому він надається.

У мовах, спеціально призначених для символьних обчислень, таких як Mathematica, якщо Ви використовуєте змінну, якої нічого не було присвоєно, вона автоматично сприймається як символ з тим же ім'ям. Пітон був спочатку призначений для символьних обчислень. Якщо Ви використовуєте змінну, якій нічого не було надано, Ви отримаєте повідомлення про помилку. Об'єкти типу Symbol слід створювати явно.

```
x=Symbol('x')
```

```
a=x**2-1
a
```

$$x^2 - 1$$

```
type(a)
```

```
sympy.core.add.Add
```

Можна визначити кілька символів одночасно. Рядок розбивається на імена за пробілами.

```
y,z=symbols('y z')
```

Підставимо замість x вираз y+1

```
a.subs(x,y+1)
```

$$(y+1)^2-1$$

Багаточлени та раціональні функції

`SymPy` не розкриває дужки автоматично. Для цього використовується функція `expand`.

```
a=(x+y-z)**6
a
```

$$(x+y-z)^6$$

```
a=expand(a)
a
```

$$x^6+6x^5y-6x^5z+15x^4y^2-30x^4yz+15x^4z^2+20x^3y^3-60x^3y^2z+60x^3yz^2-20x^3z^3+15x^2y^4-60x^2y^3z+90x^2y^2z^2-60x^2yz^3+15x^2z^4+6xy^5-30xy^4z+60xy^3z^2-60xy^2z^3+30xyz^4-6xz^5+y^6-6y^5z+15y^4z^2-20y^3z^3+15y^2z^4-6yz^5+z^6$$

Ступінь многочлена: `degree`

```
degree(a,x)
```

$$6$$

Зберемо разом члени з певними ступенями `x`: `collect`

```
collect(a,x)
```

$$x^6+x^5(6y-6z)+x^4(15y^2-30yz+15z^2)+x^3(20y^3-60y^2z+60yz^2-20z^3)+x^2(15y^4-60y^3z+90y^2z^2-60yz^3+15z^4)+x(6y^5-30y^4z+60y^3z^2-60y^2z^3+30yz^4-6z^5)+y^6-6y^5z+15y^4z^2-20y^3z^3+15y^2z^4-6yz^5+z^6$$

Багаточлен з цілими коефіцієнтами можна записати у вигляді добутку таких багаточленів (причому кожен співмножник вже неможливо розкласти на множники далі, залишаючись у рамках багаточленів з цілими коефіцієнтами).

Існують ефективні алгоритми для вирішення цієї задачі – `factor`.

```
a=factor(a)
a
```

$$(x+y-z)^6$$

Функція `simplify` намагається переписати вираз у найпростішому вигляді. Це поняття немає чіткого визначення (у різних ситуаціях найпростішими можуть вважатися різні форми висловлювання), немає алгоритму такого спрощення.

Функція `symplify` працює евристично, і неможливо передбачити, які спрощення вона спробує зробити. Тому її зручно використовувати в інтерактивних сесіях, щоб подивитися, чи вдасться їй записати вираз у якомусь розумному вигляді, але небажано використовувати у програмах. Вони краще

застосовувати більш спеціалізовані функції, які виконують одне певне перетворення висловлювання.

```
simplify(a)
```

$$\frac{x^2+y^2}{x^2-y^2}$$

Розкладання на елементарні дроби по відношенню до x та y – функція `apart`

```
apart(a,x)
```

$$-\frac{y}{x+y}+\frac{y}{x-y}+1$$

```
apart(a,y)
```

$$\frac{x}{x+y}+\frac{x}{x-y}-1$$

Підставимо конкретні чисельні значення замість змінних x та y subs

```
a=a.subs({x:1,y:2})
a
```

$$-\frac{5}{3}$$

А скільки це буде чисельно? Метод n()

```
a.n()
```

$$-1.66666666666667$$

Елементарні функції

`SymPy` автоматично застосовує спрощення елементарних функцій (які справедливі у всіх випадках).

```
sin(-x)
```

$$-\sin(x)$$

```
cos(pi/4),tan(5*pi/6)
```

$$\left(\frac{\sqrt{2}}{2}, \quad -\frac{\sqrt{3}}{3}\right)$$

`SymPy` може працювати з числами з рухомою точкою, що мають як завгодно велику точність. Ось π із 100 значущими цифрами – метод `n(100)`.

```
pi.n(100)
```

$$3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706 8$$

`E` – це основа натуральних логарифмів.

```
log(1),log(E)
```

$(0, \quad 1)$

```
exp(log(x)),log(exp(x))
```

$(x, \quad \log\left(e^{x}\right))$

```
sqrt(0)
```

0

```
sqrt(x)**4,sqrt(x**4)
```

$\left(x^{2}, \quad \sqrt{x^{4}}\right)$

Символи можуть мати деякі властивості. Наприклад, вони можуть бути позитивними. Тоді `SymPy` може сильніше спростити квадратне коріння.

```
p,q=symbols('p q',positive=True)
sqrt(p**2)
```

p

```
sqrt(12*x**2*y),sqrt(12*p**2*y)
```

$\left(2\sqrt{3}\sqrt{x^{2}y}, \quad 2\sqrt{3}p\sqrt{y}\right)$

Нехай символ `n` буде цілим (I – це уявна одиниця).

```
n=Symbol('n',integer=True)
simplify(exp(2*pi*I*n))
```

1

```
sin(pi*n)
```

0

Метод `rewrite` намагається переписати вираз у термінах заданої функції.

```
cos(x).rewrite(exp),exp(I*x).rewrite(cos)
```

$\left(\frac{e^{ix}}{2} + \frac{1}{2}e^{-ix}, \quad i\sin\left(x\right) + \cos\left(x\right)\right)$

```
asin(x).rewrite(log)
```

$-i\log\left(ix + \sqrt{-x^{2}+1}\right)$

Функція `trigsimp` намагається переписати тригонометричний вираз у найбільш простому вигляді. У програмах краще використовувати спеціалізовані функції.

```
trigsimp(2*sin(x)**2+3*cos(x)**2)
```

$\cos^{2}\left(x\right) + 2$

Функція `expand_trig` розкладає синуси та косинуси сум та кратних кутів.

```
expand_trig(sin(x-y)),expand_trig(sin(2*x))
```

$$(\sin(x)\cos(y) - \sin(y)\cos(x),\quad 2\sin(x)\cos(x))$$

Найчастіше потрібно зворотне перетворення – творів і ступенів синусів і косінусів у вирази, лінійні за цими функціями.

Наприклад, нехай ми працюємо з відрізком низки Фур'є.

```
a1,a2,b1,b2=symbols('a1 a2 b1 b2')
a=a1*cos(x)+a2*cos(2*x)+b1*sin(x)+b2*sin(2*x)
a
```

$$a_1\cos(x) + a_2\cos(2x) + b_1\sin(x) + b_2\sin(2x)$$

Ми хочемо звести його у квадрат і знову отримати відрізок ряду Фур'є.

```
a=(a**2).rewrite(exp).expand().rewrite(cos).expand()
a
```

$$\frac{a_1^2}{2}\cos(2x) + \frac{a_1^2}{2} + a_1a_2\cos(x) + a_1a_2\cos(3x) + a_1b_1\sin(2x) + a_1b_2\sin(x) + a_1b_2\sin(3x) + \frac{a_2^2}{2}\cos(4x) + \frac{a_2^2}{2} - a_2b_1\sin(x) + a_2b_1\sin(3x) + a_2b_2\sin(4x) - \frac{b_1^2}{2}\cos(2x) + \frac{b_1^2}{2} + b_1b_2\cos(x) - b_1b_2\cos(3x) - \frac{b_2^2}{2}\cos(4x) + \frac{b_2^2}{2}$$

```
a.collect([cos(x),cos(2*x),cos(3*x),sin(x),sin(2*x),sin(3*x)])
```

$$\frac{a_1^2}{2} + a_1b_1\sin(2x) + \frac{a_2^2}{2}\cos(4x) + \frac{a_2^2}{2} + a_2b_2\sin(4x) + \frac{b_1^2}{2} - \frac{b_2^2}{2}\cos(4x) + \frac{b_2^2}{2} + \left(\frac{a_1^2}{2} - \frac{b_1^2}{2}\right)\cos(2x) + (a_1a_2 - b_1b_2)\cos(3x) + (a_1a_2 + b_1b_2)\cos(x) + (a_1b_2 - a_2b_1)\sin(x) + (a_1b_2 + a_2b_1)\sin(3x)$$

Функція `expand_log` перетворює логарифми творів та ступенів у суми логарифмів (тільки для позитивних величин);

`Logcombine` здійснює зворотне перетворення.

```
a=expand_log(log(p*q**2))
a
```

$$\log(p) + 2\log(q)$$

```
logcombine(a)
```

$$\log(pq^2)$$

Функція `expand_power_exp` переписує ступеня, показники яких – суми через твори ступенів.

```
expand_power_exp(x**(p+q))
```

$$x^p x^q$$

Функція `expand_power_base` переписує ступеня, основи яких – твори, через твори ступенів.

```
expand_power_base((x*y)**n)
```

$$x^n y^n$$

Функція `powsimp` виконує зворотні перетворення.

```
powsimp(exp(x)*exp(2*y)),powsimp(x**n*y**n)
```

$$\left(e^{x+2y},\quad (xy)^n\right)$$

Можна вводити функції користувача. Вони можуть мати довільну кількість аргументів.

```
f=Function('f')
f(x)+f(x,y)
```

$$f(x)+f(x,y)$$

Структура виразів

Внутрішнє уявлення виразу – це дерево. Функція srepr повертає рядок, що його представляє.

```
srepr(x+1)
```

```
"Add(Symbol('x'), Integer(1))"
```

```
srepr(x-1)
```

```
"Add(Symbol('x'), Integer(-1))"
```

```
srepr(x-y)
```

```
"Add(Symbol('x'), Mul(Integer(-1), Symbol('y')))"
```

```
srepr(2*x*y/3)
```

```
"Mul(Rational(2, 3), Symbol('x'), Symbol('y'))"
```

```
srepr(x/y)
```

```
"Mul(Symbol('x'), Pow(Symbol('y'), Integer(-1)))"
```

Замість бінарних операцій +, *, ** та інш. можна використовувати функції `Add`, `Mul`, `Pow` тощо.

```
Mul(x,Pow(y,-1))==x/y
```

```
True
```

```
srepr(f(x,y))
```

```
"Function('f')(Symbol('x'), Symbol('y'))"
```

Атрибут `func` – це функція верхнього рівня у виразі, a args – список її аргументів.

```
a=2*x*y**2
a.func
```

```
sympy.core.mul.Mul
```

```
a.args
```

$$(2,\quad x,\quad y^2)$$

```
a.args[0]
```

$$2$$

```
for i in a.args:
    print(i)
```

```
2
x
y**2
```

Функція subs замінює змінну вираз.

```
a.subs(y,2)
```

$$8x$$

Вона може замінити кілька змінних. Для цього їй передається список кортежів чи словник.

```
a.subs([(x,pi),(y,2)])
```

$$8\pi$$

```
a.subs({x:pi,y:2})
```

$$8\pi$$

Вона може замінити не змінну, а підвираз – функцію з аргументами.

```
a=f(x)+f(y)
a.subs(f(y),1)
```

$$f(x)+1$$

```
(2*x*y*z).subs(x*y,z)
```

$$2z^2$$

```
(x+x**2+x**3+x**4).subs(x**2,y)
```

$$x^3+x+y^2+y$$

Підстановки виконуються послідовно. У разі спочатку x замінився на y, вийшло y3+y2; потім у цьому результаті y замінилося на x

```
a=x**2+y**3
a.subs([(x,y),(y,x)])
```

$$x^3+x^2$$

Якщо написати ці підстановки в іншому порядку, результат буде іншим.

```
a.subs([(y,x),(x,y)])
```

$$y^3 + y^2$$

Але можна передати функції subs ключовий параметр `simultaneous = True`, тоді підстановки будуть виконуватися одночасно. Таким чином можна, наприклад, поміняти місцями x та y

```
a.subs([(x,y),(y,x)],simultaneous=True)
```

$$x^3 + y^2$$

Можна замінити функцію іншою функцією.

```
g=Function('g')
a=f(x)+f(y)
a.subs(f,g)
```

$$g(x) + g(y)$$

Метод `replace` шукає підвирази, відповідні зразку (який містить довільні змінні), і замінює їх у заданий вираз (воно може містити самі довільні змінні).

```
a=Wild('a')
(f(x)+f(x+y)).replace(f(a),a**2)
```

$$x^2 + (x+y)^2$$

```
(f(x,x)+f(x,y)).replace(f(a,a),a**2)
```

$$x^2 + f(x,y)$$

```
a=x**2+y**2
a.replace(x,x+1)
```

$$y^2 + (x+1)^2$$

Відповідати зразку має ціле підвираз, це може бути частина співмножників у творі чи менша ступінь більшою.

```
a=2*x*y*z
a.replace(x*y,z)
```

$$2xyz$$

```
(x+x**2+x**3+x**4).replace(x**2,y)
```

$$x^4 + x^3 + x + y$$

Розв'язання рівнянь

```
a,b,c,d,e,f=symbols('a b c d e f')
```

Рівняння записується як функція `Eq` із двома параметрами. Функція `solve` повертає перелік рішень.

```
solve(Eq(a*x,b),x)
```

$$\left[\frac{b}{a}\right]$$

Втім, можна передати функції `solve` просто вираз. Мається на увазі рівняння, що цей вираз дорівнює 0.

```
solve(a*x+b,x)
```

$$\left[-\frac{b}{a}\right]$$

Квадратне рівняння має два рішення.

```
solve(a*x**2+b*x+c,x)
```

$$\left[\frac{1}{2a}\left(-b+\sqrt{-4ac+b^2}\right),\quad -\frac{1}{2a}\left(b+\sqrt{-4ac+b^2}\right)\right]$$

Система лінійних рівнянь

```
solve([a*x+b*y-e,c*x+d*y-f],[x,y])
```

$$\left\{x:\frac{-bf+de}{ad-bc},\quad y:\frac{af-ce}{ad-bc}\right\}$$

Функція `roots` повертає коріння багаточлена зі своїми кратностями.

```
roots(x**3-3*x+2,x)
```

$$\{-2:1,\quad 1:2\}$$

Функція `solve_poly_system` вирішує систему поліноміальних рівнянь, будуючи їх базис Гребнера.

```
p1=x**2+y**2-1
p2=4*x*y-1
solve_poly_system([p1,p2],x,y)
```

$$\left[\left(4\left(-1-\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}\right)\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}\left(-\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}+1\right),\quad -\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}\right),\right.$$
$$\left(-4\left(-1+\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}\right)\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}\left(\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}+1\right),\quad \sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}\right),$$
$$\left(4\left(-1-\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}\right)\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}\left(-\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}+1\right),\quad -\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}\right),$$
$$\left.\left(-4\left(-1+\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}\right)\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}\left(\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}+1\right),\quad \sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}\right)\right]$$

Ряди

```
exp(x).series(x,0,5)
```

$$1+x+\frac{x^2}{2}+\frac{x^3}{6}+\frac{x^4}{24}+\mathcal{O}\left(x^5\right)$$

Ряд може починатися з негативного ступеня.

```
cot(x).series(x,n=5)
```

$$\frac{1}{x} - \frac{x}{3} - \frac{x^3}{45} + \mathcal{O}\left(x^5\right)$$

```
sqrt(x*(1-x)).series(x,n=5)
```

$$\sqrt{x} - \frac{x^{\frac{3}{2}}}{2} - \frac{x^{\frac{5}{2}}}{8} - \frac{x^{\frac{7}{2}}}{16} - \frac{5x^{\frac{9}{2}}}{128} + \mathcal{O}\left(x^5\right)$$

```
log(gamma(1+x)).series(x,n=6).rewrite(zeta)
```

$$-\gamma x + \frac{\pi^2 x^2}{12} - \frac{x^3\zeta\left(3\right)}{3} + \frac{\pi^4 x^4}{360} - \frac{x^5\zeta\left(5\right)}{5} + \mathcal{O}\left(x^6\right)$$

Підготуємо 3 ряди.

```
sinx=series(sin(x),x,0,8)
sinx
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \mathcal{O}\left(x^8\right)$$

```
cosx=series(cos(x),x,n=8)
cosx
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \mathcal{O}\left(x^8\right)$$

```
tanx=series(tan(x),x,n=8)
tanx
```

$$x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \mathcal{O}\left(x^8\right)$$

Ряди можна диференціювати та інтегрувати.

```
diff(sinx,x)
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \mathcal{O}\left(x^7\right)$$

```
integrate(cosx,x)
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \mathcal{O}\left(x^9\right)$$

Можна підставити ряд (якщо він починається з малого члена) замість змінного розкладання до іншого ряду. Ось ряди для `sin(tan(x))` та `tan(sin(x))`

```
st=series(sinx.subs(x,tanx),n=8)
st
```

$$x + \frac{x^3}{6} - \frac{x^5}{40} - \frac{55x^7}{1008} + \mathcal{O}\left(x^8\right)$$

```
ts=series(tanx.subs(x,sinx),n=8)
ts
```

$$x + \frac{x^3}{6} - \frac{x^5}{40} - \frac{107x^7}{5040} + \mathcal{O}\left(x^8\right)$$

```
series(ts-st,n=8)
```

$$\frac{x^7}{30} + \mathcal{O}\left(x^8\right)$$

Похідні

```
a=x*sin(x+y)
diff(a,x)
```

$$x\cos(x+y)+\sin(x+y)$$

```
diff(a,y)
```

$$x\cos(x+y)$$

Друга похідна x і перша y:

```
diff(a,x,2,y)
```

$$-x\cos(x+y)+2\sin(x+y)$$

Диференціювання виразів, що містять певні функції:

```
a=x*f(x**2)
b=diff(a,x)
b
```

$$2x^2\left.\frac{d}{d\xi_1}f(\xi_1)\right|_{\xi_1=x^2}+f(x^2)$$

```
print(b)
```

```
2*x**2*Subs(Derivative(f(_xi_1), _xi_1), (_xi_1,), (x**2,)) + f(x**2)
```

Функція `Derivative` є не обчисленою похідною. Її можна визначити методом doit.

```
a=Derivative(sin(x),x)
Eq(a,a.doit())
```

$$\frac{d}{dx}\sin(x)=\cos(x)$$

Інтеграли

```
integrate(1/(x*(x**2-2)**2),x)
```

$$\frac{1}{4}\log(x)-\frac{1}{8}\log(x^2-2)-\frac{1}{4x^2-8}$$

```
integrate(1/(exp(x)+1),x)
```

$$x-\log(e^x+1)$$

```
integrate(log(x),x)
```

$$x\log(x)-x$$

```
integrate(x*sin(x),x)
```

$$-x\cos(x)+\sin(x)$$

```
integrate(x*exp(-x**2),x)
```

$$-\frac{e^{-x^2}}{2}$$

```
a=integrate(x**x,x)
a
```

$$\int x^x\,dx$$

Невизначений інтеграл:

```
print(a)
```

```
Integral(x**x, x)
```

```
a=Integral(sin(x),x)
Eq(a,a.doit())
```

$$\int \sin(x)\,dx = -\cos(x)$$

Визначені інтеграли.

```
integrate(sin(x),(x,0,pi))
```

$$2$$

oo - это ∞.

```
integrate(exp(-x**2),(x,0,oo))
```

$$\frac{\sqrt{\pi}}{2}$$

```
integrate(log(x)/(1-x),(x,0,1))
```

$$-\frac{\pi^2}{6}$$

Суми рядів

```
summation(1/n**2,(n,1,oo))
```

$$\frac{\pi^2}{6}$$

```
summation((-1)**n/n**2,(n,1,oo))
```

$$-\frac{\pi^2}{12}$$

```
summation(1/n**4,(n,1,oo))
```

$$\frac{\pi^4}{90}$$

```
a=Sum(x**n/factorial(n),(n,0,oo))
Eq(a,a.doit())
```

$$\sum_{n=0}^{\infty} \frac{x^n}{n!} = e^x$$

Межі

```
limit((tan(sin(x))-sin(tan(x)))/x**7,x,0)
```

$$\frac{1}{30}$$

Це проста межа, вона вважається розкладанням чисельника та знаменника в ряди. А якщо x=0, то з'являється особлива точка.

Знайдемо односторонні межі.

```
limit((tan(sin(x))-sin(tan(x)))/(x**7+exp(-1/x)),x,0,'+')
```

$$\frac{1}{30}$$

```
limit((tan(sin(x))-sin(tan(x)))/(x**7+exp(-1/x)),x,0,'-')
```

$$0$$

Диференціальні рівняння

```
t=Symbol('t')
x=Function('x')
p=Function('p')
```

Перший порядок.

```
dsolve(diff(x(t),t)+x(t),x(t))
```

$$x(t) = C_1 e^{-t}$$

Другий порядок.

```
dsolve(diff(x(t),t,2)+x(t),x(t))
```

$$x(t) = C_1 \sin(t) + C_2 \cos(t)$$

Система рівнянь першого ладу.

```
dsolve((diff(x(t),t)-p(t),diff(p(t),t)+x(t)))
```

$$[x(t) = C_1 \sin(t) + C_2 \cos(t), \quad p(t) = C_1 \cos(t) - C_2 \sin(t)]$$

Лінійна алгебра

```
a,b,c,d,e,f=symbols('a b c d e f')
```

Матрицю можна побудувати зі списку списків.

```
M=Matrix([[a,b,c],[d,e,f]])
M
```

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

```
M.shape
```

$$(2,\ 3)$$

Матриця рядок.

```
Matrix([[1,2,3]])
```

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

Матриця стовпець.

```
Matrix([1,2,3])
```

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Можна побудувати матрицю із функції.

```
def g(i,j):
    return Rational(1,i+j+1)
Matrix(3,3,g)
```

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

Або з невизначеної функції.

```
g=Function('g')
M=Matrix(3,3,g)
M
```

$$\begin{bmatrix} g(0,0) & g(0,1) & g(0,2) \\ g(1,0) & g(1,1) & g(1,2) \\ g(2,0) & g(2,1) & g(2,2) \end{bmatrix}$$

```
M[1,2]
```

$$g(1,2)$$

```
M[1,2]=0
M
```

$$\begin{bmatrix} g(0,0) & g(0,1) & g(0,2) \\ g(1,0) & g(1,1) & 0 \\ g(2,0) & g(2,1) & g(2,2) \end{bmatrix}$$

```
M[2,:]
```

$$\begin{bmatrix} g(2,0) & g(2,1) & g(2,2) \end{bmatrix}$$

```
M[:,1]
```

$$\begin{bmatrix} g(0,1) \\ g(1,1) \\ g(2,1) \end{bmatrix}$$

```
M[0:2,1:3]
```

$$\begin{bmatrix} g(0,1) & g(0,2) \\ g(1,1) & 0 \end{bmatrix}$$

Одинична матриця.

```
eye(3)
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Матриця з нулів.

```
zeros(3)
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

```
zeros(2,3)
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Діагональна матриця.

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

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

```
M=Matrix([[a,1],[0,a]])
diag(1,M,2)
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & a & 1 & 0 \\ 0 & 0 & a & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Операції із матрицями.

```
A=Matrix([[a,b],[c,d]])
B=Matrix([[1,2],[3,4]])
A+B
```

$$\begin{bmatrix} a+1 & b+2 \\ c+3 & d+4 \end{bmatrix}$$

```
A*B,B*A
```

$$\left(\begin{bmatrix} a+3b & 2a+4b \\ c+3d & 2c+4d \end{bmatrix}, \begin{bmatrix} a+2c & b+2d \\ 3a+4c & 3b+4d \end{bmatrix}\right)$$

```
A*B-B*A
```

$$\begin{bmatrix} 3b-2c & 2a+3b-2d \\ -3a-3c+3d & -3b+2c \end{bmatrix}$$

```
simplify(A**(-1))
```

$$\begin{bmatrix} \frac{d}{ad-bc} & -\frac{b}{ad-bc} \\ -\frac{c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}$$

```
det(A)
```

$$ad - bc$$

Власні значення та вектори

```
x=Symbol('x',real=True)
```

```
M=Matrix([[(1-x)**3*(3+x),4*x*(1-x**2),-2*(1-x**2)*(3-x)],
          [4*x*(1-x**2),-(1+x)**3*(3-x),2*(1-x**2)*(3+x)],
          [-2*(1-x**2)*(3-x),2*(1-x**2)*(3+x),16*x]])
M
```

$$\begin{bmatrix} (-x+1)^3(x+3) & 4x\left(-x^2+1\right) & (-x+3)\left(2x^2-2\right) \\ 4x\left(-x^2+1\right) & -(-x+3)(x+1)^3 & (x+3)\left(-2x^2+2\right) \\ (-x+3)\left(2x^2-2\right) & (x+3)\left(-2x^2+2\right) & 16x \end{bmatrix}$$

```
det(M)
```

$$0$$

Значить, ця матриця має нульовий підпростір (вона звертає вектори з цього підпростору в 0). Базис цього підпростору.

```
v=M.nullspace()
len(v)
```

$$1$$

Він одномірний:

```
v=simplify(v[0])
v
```

$$\begin{bmatrix} -\frac{2}{x-1} \\ \frac{2}{x+1} \\ 1 \end{bmatrix}$$

Перевіримо.

```
simplify(M*v)
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Власні значення та його кратності.

```
M.eigenvals()
```

$$\left\{0 : 1, \quad -\left(x^2+3\right)^2 : 1, \quad \left(x^2+3\right)^2 : 1\right\}$$

Якщо потрібні не лише власні значення, а й власні вектори, то потрібно використовувати метод eigenvects. Він повертає список кортежів. У кожному їх нульовий елемент – власне значення, перший – його кратність, і останній – перелік власних векторів, що утворюють базис (їх стільки, яка кратність).

```
v=M.eigenvects()
len(v)
```

3

```
for i in range(len(v)):
    v[i][2][0]=simplify(v[i][2][0])
v
```

$$\left[\left(0,\quad 1,\quad \left[\begin{bmatrix} -\frac{2}{x-1} \\ \frac{2}{x+1} \\ 1 \end{bmatrix}\right]\right),\quad \left(-\left(x^2+3\right)^2,\quad 1,\quad \left[\begin{bmatrix} \frac{x}{2}+\frac{1}{2} \\ \frac{x+1}{x-1} \\ 1 \end{bmatrix}\right]\right),\quad \left(\left(x^2+3\right)^2,\quad 1,\quad \left[\begin{bmatrix} \frac{x-1}{x+1} \\ -\frac{x}{2}+\frac{1}{2} \\ 1 \end{bmatrix}\right]\right)\right]$$

Перевіримо.

```
for i in range(len(v)):
    z=M*v[i][2][0]-v[i][0]*v[i][2][0]
    pprint(simplify(z))
```

```
⎡0⎤
⎢ ⎥
⎢0⎥
⎢ ⎥
⎢0⎥
⎡0⎤
⎢ ⎥
⎢0⎥
⎢ ⎥
⎢0⎥
⎡0⎤
⎢ ⎥
⎢0⎥
⎢ ⎥
⎢0⎥
```

Жорданова нормальна форма

```
M=Matrix([[Rational(13,9),-Rational(2,9),Rational(1,3),Rational(4,9),Rational(2,3)],
          [-Rational(2,9),Rational(10,9),Rational(2,15),-Rational(2,9),-Rational(11,15)],
          [Rational(1,5),-Rational(2,5),Rational(41,25),-Rational(2,5),Rational(12,25)],
          [Rational(4,9),-Rational(2,9),Rational(14,15),Rational(13,9),-Rational(2,15)],
          [-Rational(4,15),Rational(8,15),Rational(12,25),Rational(8,15),Rational(34,25)]])
M
```

$$\begin{bmatrix} \frac{13}{9} & -\frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \frac{2}{3} \\ -\frac{2}{9} & \frac{10}{9} & \frac{2}{15} & -\frac{2}{9} & -\frac{11}{15} \\ \frac{1}{5} & -\frac{2}{5} & \frac{41}{25} & -\frac{2}{5} & \frac{12}{25} \\ \frac{4}{9} & -\frac{2}{9} & \frac{14}{15} & \frac{13}{9} & -\frac{2}{15} \\ -\frac{4}{15} & \frac{8}{15} & \frac{12}{25} & \frac{8}{15} & \frac{34}{25} \end{bmatrix}$$

Метод M.jordan_form() повертає пару матриць, матрицю перетворення P та власне жорданову форму J: M=PJP−1

```
P,J=M.jordan_form()
J
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1-i & 0 \\ 0 & 0 & 0 & 0 & 1+i \end{bmatrix}$$

```
P=simplify(P)
P
```

$$\begin{bmatrix} -2 & \frac{10}{9} & 0 & \frac{5i}{12} & -\frac{5i}{12} \\ -2 & -\frac{5}{9} & 0 & -\frac{5i}{6} & \frac{5i}{6} \\ 0 & 0 & \frac{4}{3} & -\frac{3}{4} & -\frac{3}{4} \\ 1 & \frac{10}{9} & 0 & -\frac{5i}{6} & \frac{5i}{6} \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Перевіримо.

```
Z=P*J*P**(-1)-M
simplify(Z)
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

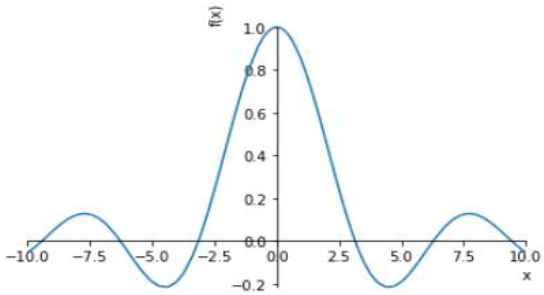
Графіки

SymPy використовує matplotlib. Однак він розподіляє точки по x адаптивно, а не рівномірно.

```
%matplotlib inline
```

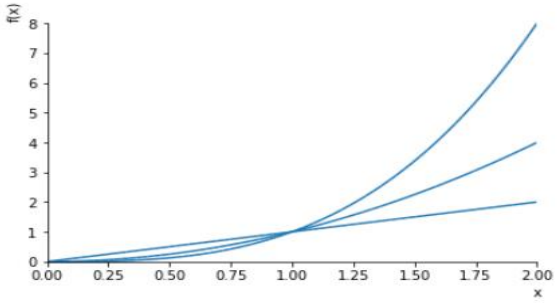
Одна функція.

```
plot(sin(x)/x,(x,-10,10))
```



Декілька функцій.

```
plot(x,x**2,x**3,(x,0,2))
```

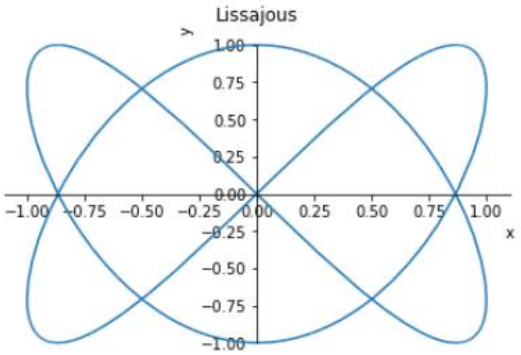


Інші функції слід імпортувати з пакету `sympy.plotting`.

```
from sympy.plotting import (plot_parametric,plot_implicit,
     plot3d,plot3d_parametric_line, plot3d_parametric_surface)
```

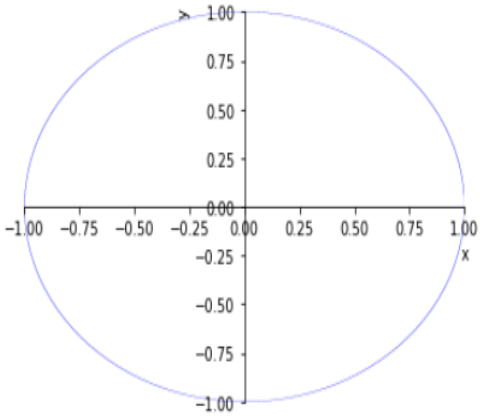
Параметричний графік– фігура Ліссажу.

```
t=Symbol('t')

plot_parametric(sin(2*t),cos(3*t),(t,0,2*pi),
 title='Lissajous',xlabel='x',ylabel='y')
```



Неявний графік – Коло.

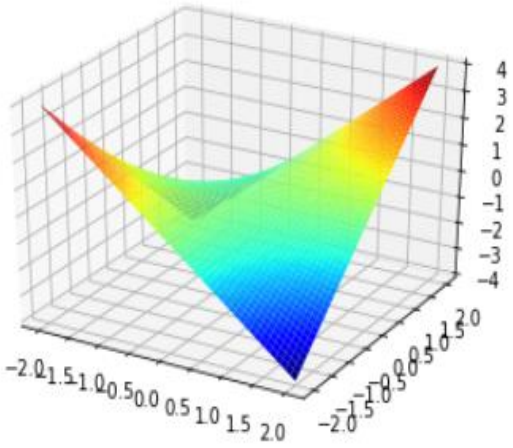
```
plot_implicit(x**2+y**2-1,(x,-1,1),(y,-1,1))
```



Поверхня.

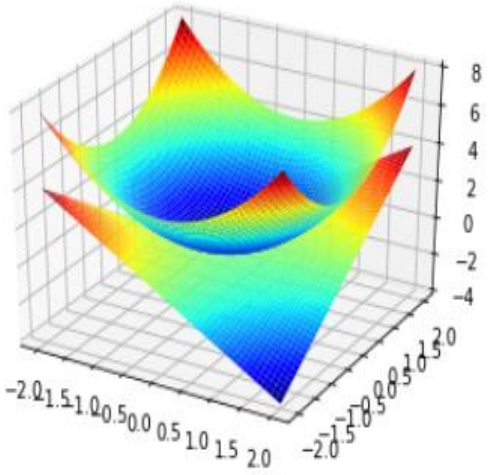
Якщо вона будується не inline, а в окремому вікні, її можна крутити мишкою.

```
plot3d(x*y,(x,-2,2),(y,-2,2))
```



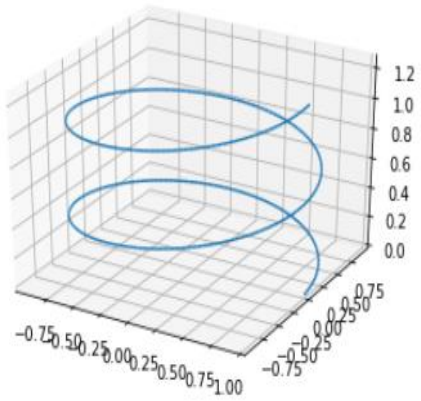
Декілька поверхонь.

```
plot3d(x**2+y**2,x*y,(x,-2,2),(y,-2,2))
```



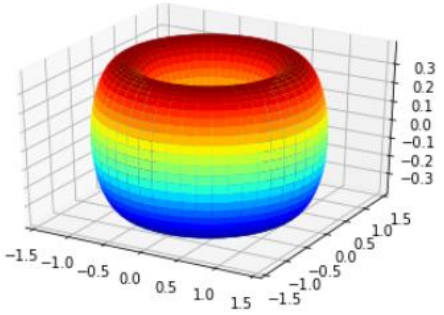
Параметрична просторова лінія – Спіраль.

```
a=0.1
plot3d_parametric_line(cos(t),sin(t),a*t,(t,0,4*pi))
```



Параметрична поверхня – тор.

```
u,v=symbols('u v')
a=0.4
plot3d_parametric_surface((1+a*cos(u))*cos(v),
(1+a*cos(u))*sin(v),a*sin(u),
(u, 0,2 * pi), (v, 0,2 * pi))
```



Питання для самоконтролю до теми 30

1. Що таке «символьні обчислення»?
2. Яке основне призначення бібліотеки SymPy?
3. Чи можна використовувати SymPy як калькулятор?
4. Як задаються (визначаються) змінні до виконання «символьних обчислень»?
5. Які математичні процедури можна виконати за допомогою бібліотечних методів бібліотеки SymPy?

Завдання до теми 30.

Розробите програму, яка вводить текстовий рядок, що містить запис функціональної залежності f(x). Програма з використанням бібліотеки SymPy має знайти та вивести на консоль похідну та інтеграл від f(x).

ВИКОРИСТАНА ЛІТЕРАТУРА

1. Unseen Perspectives. Python in Practice: From Fundamentals to Functional Code: Your essential first step into coding, Leanpub, 2025, 120 p.
2. Stepic R. Python GUI with Tkinter – From Beginner to Pro. Leanpub/STREM Academy, 2025. 601 p.
3. Oliveira W. Python for Beginners: Mastering the Basics of Python. Independently published, 2025. 500 p.
4. Hellmann D. Python 3 Standard Library by Example. 2nd ed., 2022. 1456 p.
5. Горобець О. Ю., Горобець С. В., Хахно К. Ю. Мова Python для інженерних та наукових задач : підручник. Київ : КПІ ім. Ігоря Сікорського, 2024. 277 с.
6. Копей В. Б. Мова програмування Python для інженерів і науковців : навч. посіб. Івано-Франківськ : ІФНТУНГ, 2019. 272 с.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Горобець О. Ю., Горобець С. В., Хахно К. Ю. Мова Python для інженерних та наукових задач : підручник. Київ : КПІ ім. Ігоря Сікорського, 2024. 277 с.
2. Копей В. Б. Мова програмування Python для інженерів і науковців : навч. посіб. Івано-Франківськ : ІФНТУНГ, 2019. 272 с.
3. Sach L., O'Hanlon M. Create Graphical User Interfaces with Python: How to build windows, buttons, and widgets for your Python projects. Cambridge : Raspberry Pi Trading, 2023. 156 p

Інформаційні ресурси

1. NumPy Documentation. *NumPy*. URL: https://numpy.org/doc/.
2. Welcome to Python. *Python.org*. URL: https://www.python.org/doc/.
3. Visualization with Python. *MATPLOTLIB*. URL: https://matplotlib.org/
4. Numpy and Scipy Documentation. *SciPy.org*. URL: https://docs.scipy.org/doc/
5. Why SymPy. *SymPy*. URL: http://www.sympy.org/en/index.html

ЗМІСТ

1. МОДУЛІ ТА ПАКЕТИ В PYTHON5
5
5
6
6
7
7
8
8
9
9
10
10
10
11
2. БІБЛІОТЕКА NUMPY12
12
12
12
12
16
17
18
23
25
27
27
28
28
29
29
30
31
31
32
`linalg` деякі операції лінійної алгебри33
34
Винятки `numpy.linalg.LinAlgError`34
35
38
42
43
3. БІБЛІОТЕКА MATPLOTLIB44
44
45
48
50
Список координат, x координати числа 0, 1, 2, 3...51
Списки x та y координат51
Зображення точок51

«Довільні» координати51
Декілька графіків на одному аркуші52
«Прикрашання» та багато графіків на діаграмі52
Два графіки на діаграмі в індивідуальних масштабах56
58
61
63
63
65
66
67
71
71
73
73
74
75
77
78
79
82
84
85
87
91
92
94
95
96
99
100
101
101
4. МОДУЛЬ MATH102
103
103
5. МОДУЛЬ `fractions`: РОБОТА ІЗ РАЦІОНАЛЬНИМИ ЧИСЛАМИ104
104
106
`Fraction` – атрибути та методи106
110
110
110
6. МОДУЛЬ `cmath`111
112
112
7. МОДУЛЬ `struct` – ЗАПИС ДАНИХ У БІНАРНИЙ ФАЙЛ113
113
114
114
115
116

116
8. ФАЙЛИ CSV117
117
117
117
119
122
123
123
9. МОДУЛЬ `shelve` (*.INI)124
125
126
126
126
10. МОДУЛЬ `OS` – РОБОТА З ФАЙЛОВОЮ СИСТЕМОЮ127
127
127
127
128
128
130
130
11. ПРИКЛАД КОМАНД РОБОТИ З ФАЙЛАМИ ТА ФАЙЛОВОЮ СИСТЕМОЮ132
132
132
133
133
134
135
136
137
138
138
12. МОДУЛЬ `shutil`139
139
141
142
142
13. ПРИКЛАДИ ВИКОРИСТАННЯ МОДУЛЯ `SHUTIL`144
144
144
145
145
Приклад реалізації функції `shutil.copytree()` 146
147
147
147
14. Модуль `pathlib`147
150
150
151
152

153
153
15. Модуль glob154
155
155
16. Модуль os.path156
157
157
17. Модуль sys158
160
160
18. Модуль itertools161
162
19. МОДУЛЬ locale162
165
165
20. МОДУЛЬ datetime166
Клас date166
Клас time166
Клас datetime167
168
168
169
Властивості timedelta170
171
171
171
21. МОДУЛЬ logging172
173
173
22. СТВОРЕННЯ GUI ЗА ДОПОМОГОЮ БІБЛІОТЕКИ tkinter174
Введення в tkinter174
Імпорт модуля tkinter174
174
175
175
175
176
176
177
Метод place() абсолютне позиціонування177
179
179
181
Розмітка grid() у tkinter для створення калькулятора183
185
188
188
23. Віджети (графічні об'єкти) та їх властивості.189
189
189

189
Методи Entry191
193
193
193
194
194
Frame (рамка)194
195
195
Toplevel (вікно верхнього рівня)196
196
196
24. ПРОГРАМУВАННЯ ПОДІЙ197
197
199
199
199
199
203
203
25. ЗМІННІ tkinter204
206
206
26. Об'єкт меню в GUI.207
207
208
208
209
209
27. ДІАЛОГОВІ ВІКНА В tkinter210
212
212
28. ГЕОМЕТРИЧНІ ПРИМІТИВИ CANVAS (ПОЛОТНО)213
215
218
221
221
29. ДЕКІЛЬКА ПРИКЛАДІВ222
222
226
233
233
30. СИМВОЛЬНІ ОБЧИСЛЕННЯ МОВОЮ PYTHON.234
Математична бібліотека Python SymPy234
Установка SymPy234
234
236
237
238
238
239

240
241
241
242
243
243
243
244
245
245
245
246
249
250
251
254
256
257
257
259
259
260
261
261
261
266
269
269

Ошибка! Закладка не определена.

Навчальне видання
(*українською мовою*)

Борю Сергій Юрійович
Кревсун Юрій Миколайович
Миронова Наталя Олексіївна

ПРОГРАМУВАННЯ: ПОПУЛЯРНІ МОДУЛІ ТА ПАКЕТИ В PYTHON

Навчальний посібник
для здобувачів ступеня вищої освіти бакалавра
спеціальності «Комп’ютерні науки»
освітньо-професійної програми «Комп’ютерні науки»

Рецензент *С. І. Гоменюк*
Відповідальний за випуск *О. С. Пшенична*
Коректор *Н. О. Миронова*