

Міністерство освіти і науки України
Національний університет
«Полтавська політехніка імені Юрія Кондратюка»
Кафедра комп'ютерних та інформаційних технологій і систем

«PYTHON – МОВА ПРОГРАМУВАННЯ НАДВИСОКОГО РІВНЯ»

Елементи теорії і практики із структурного програмування і
сценарії з машинного навчання

Лабораторний практикум і методичні вказівки з дисципліни
«Машинне навчання»

для магістрів спеціальності 122 – комп'ютерні науки



Полтава 2022

«Python – мова програмування надвисокого рівня». Елементи теорії, практикум із структурного програмування та сценарії з машинного навчання. Лабораторний практикум і методичні вказівки з дисципліни «Машинне навчання» для магістрів спеціальності 122 – комп’ютерні науки. – Полтава: ПолтНТУ, 2022. – 126 с.

Укладач: доктор техн. наук, професор О.Л. Ляхов

Відповідальний за випуск: доцент кафедри комп’ютерних та інформаційних технологій і систем А.М. Капітон, д. п. н., доцент

Рецензент: В.В. Васюта, кандидат техн. наук., доцент

Затверджено науково-методичною радою університету

Протокол № ____ від _____ 2022 р.

Авторська редакція

Дисципліна «**Машинне навчання**» – одна з основ для становлення студента, як фахівця-практика у сучасних і перспективних галузях IT: ML, AI, Data Science, Data Mining, BI, BA і т.і.

Даний посібник – спроба вирішити, в умовах відносно невеликого часу, непросту задачу – викласти мінімум, достатній для магістра. По-перше, це елементи структурного програмування мовою надвисокого рівня (VHLL), що непередбачена базовою підготовкою студентів за фахом 122. По-друге, показати, як можна, за допомогою цього інструментарію, вирішувати базові задачі машинного навчання.

Уявляється, що найбільш прийнятний підхід у таких умовах, це навчання на прикладах. У якості основних інформаційних ресурсів студентам пропонується класична книга із штучного інтелекту і відмінна, на наш погляд, книга з мови **Python**, у якій використовується саме такий підхід для навчання тих, хто дійсно хоче стати фахівцем у машинному навчанні.

Примітка: Таблиці та рисунки у тексті не мають наскрізної нумерації, оскільки немає посилань за межі будь-якої лабораторної роботи.

Лабораторна робота № 1

Тема: «Суперкомп'ютери, мови програмування та транслятори»

Мета: Встановити місце мови Python у сучасному прикладному програмуванні

Теоретичний мінімум

Суперкомп'ютери

1. Наведіть визначення суперкомп'ютера і невелику довідку про їх призначення і типове застосування.
2. Перейдіть на сайт **TOP500 Supercomputer Sites** і відкрийте список суперкомп'ютерів (меню *List -> [Перший пункт]*), на основі якого зберіть інформацію про 10 суперкомп'ютерів:
 - № п/п;
 - назва;
 - рік першого запуска;
 - країна;
 - виробник;
 - швидкодія;
 - кількість ядер;
 - споживна потужність;
 - пам'ять;
 - зображення.
3. Наприкінці порівняйте між собою швидкодію:
 - суперкомп'ютера;
 - першої OEM Марк I;
 - Вашого мобільного пристроя.

Мови програмування сьогодні

1. Відкрийте **рейтинг мов програмування від IEEE**.
2. Вкажіть, на основі чого будується даний індекс популярності і заповніть наступну таблицю:

Тип ПЗ	Web	Mobile	Enterprise	Embedded
Місце у рейтингу				
1				

2				
3				
4				
5				

За результатами заповнення таблиці дайте відповідь на запитання:

- яка мова програмування є найбільш універсальною (охоплює більше типів ПЗ)?
- яка мова програмування найменш універсальна?

Тип транслятору

На сайті TIOBE є власний **рейтинг популярності мов**.

Вкажіть, на основі чого будується даний індекс популярності, після чого візьміть перші 15 мов програмування і заповніть наступну таблицю, використовуючи інформацію із мережі Інтернет:

№	Мова	Тип транслятору
1	Java	Гібрид
.	.	.

Після заповнення:

- визначте кількість мов програмування з вибраним типом транслятору (наприклад, що компілюються – 6, гібридів – 2 і т. і.);
- Сформулюйте власну думку, чим можна пояснити таке відношення.

Контрольні питання

- Наведіть приклади програм, які зустрічаються Вам у житті (не тільки у рамках персонального комп’ютера дома).
- Що розуміють під терміном «програмне забезпечення» (ПЗ)?
- З яких етапів складається розробка ПЗ сьогодні?
- Дайте визначення поняття «алгоритм», «програма» і «мова програмування».
- Вкажіть способи представлення і властивості алгоритму. Наведіть приклад (або анти приклад) алгоритмів для кожної з властивостей.

6. Які способи «запрограмувати» пристрій Вам відомі? Наведіть приклади.
7. Як здійснювалося програмування на перших ЕОМ?
8. Назвіть основні періоди розвитку мов високого рівня, основні особливості і найбільш яркі представники мов програмування.
9. Які класифікації мов програмування високого рівня існують? Наведіть приклади.
10. У чому відміни компіляторів від інтерпретаторів? У чому полягає смисл «гіbridної» технології? Наведіть приклади мов, які мають вказану реалізацію.
11. Система типів у мовах програмування.
12. Як обрати мову програмування? Якими основними критеріями і принципами слід керуватися?

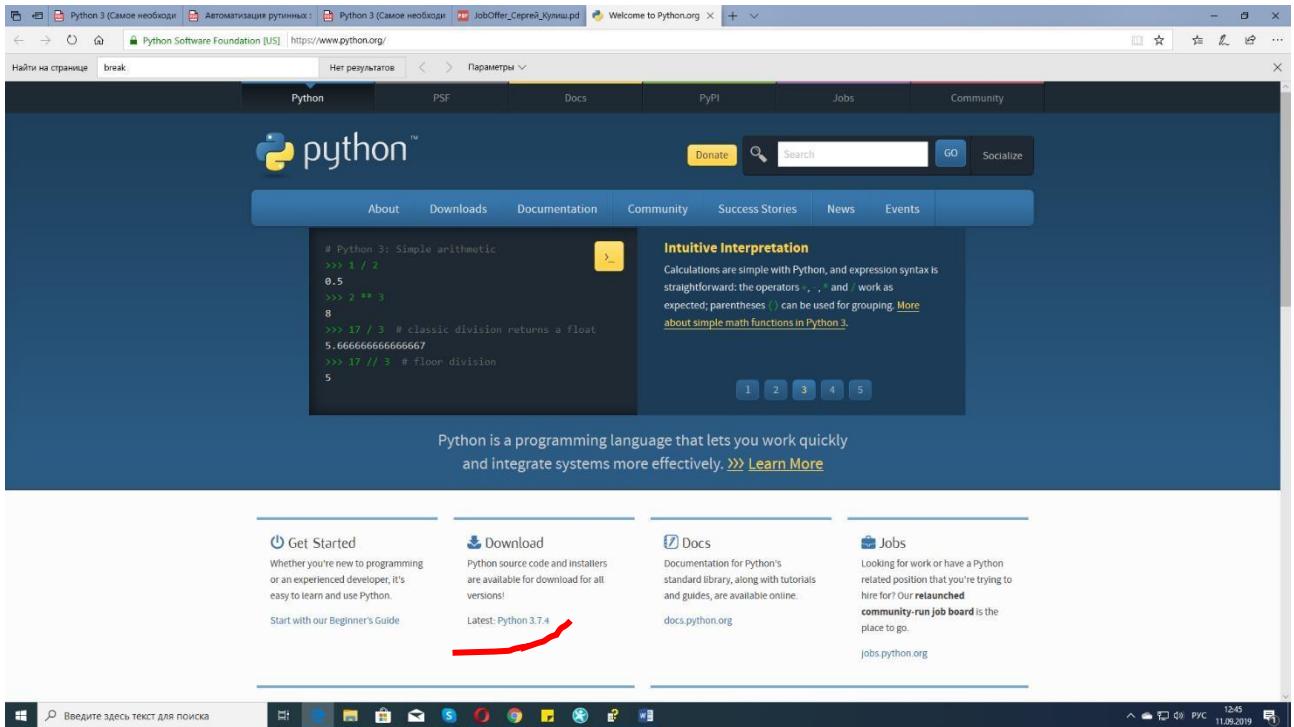
Лабораторна робота №2

Тема: «Установка Python на комп’ютер»

*Мета: Отримати навички із інсталяції Python на комп’ютер, а також, з менеджером пакетів pip **Теоретичний мінімум***

Установка Python на Windows

Скачувати **Python** варто з офіційного сайту. Використовувати інші джерела немає сенсу, оскільки ресурс безкоштовний і bezpeчний. Посилання на сайт Welcome to Python.org У стовпчику **Download** вказано посилання на останню актуальну версію

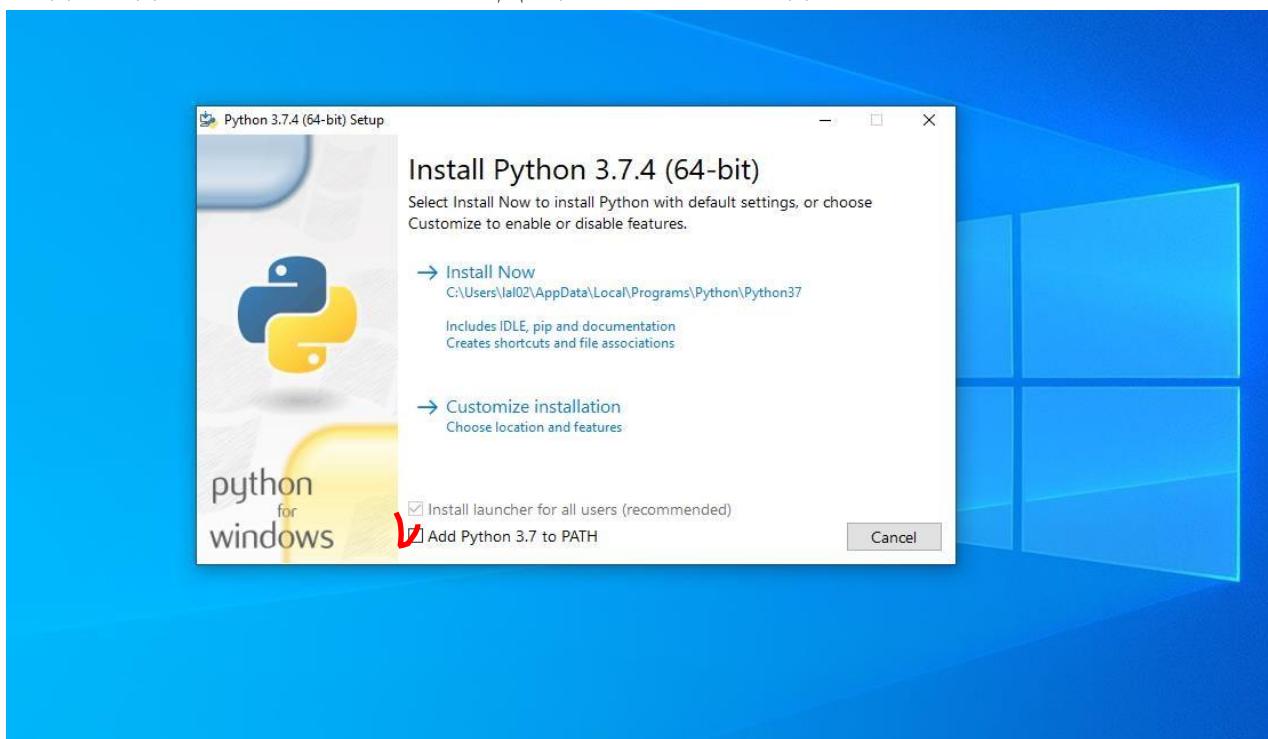


На момент роботи з посібником це було **python 3.7.4**. Заходимо за посиланням і знизу знаходимо розділ “Files”, а у ньому різні варіанти інсталятору для 32x або 64-х розрядних ОС Windows. На сьогодні це вже версія **python 3.10.x**

Якщо перейти за посиланням Download -> Windows , то відкриється перелік усіх досяжних версій Python.

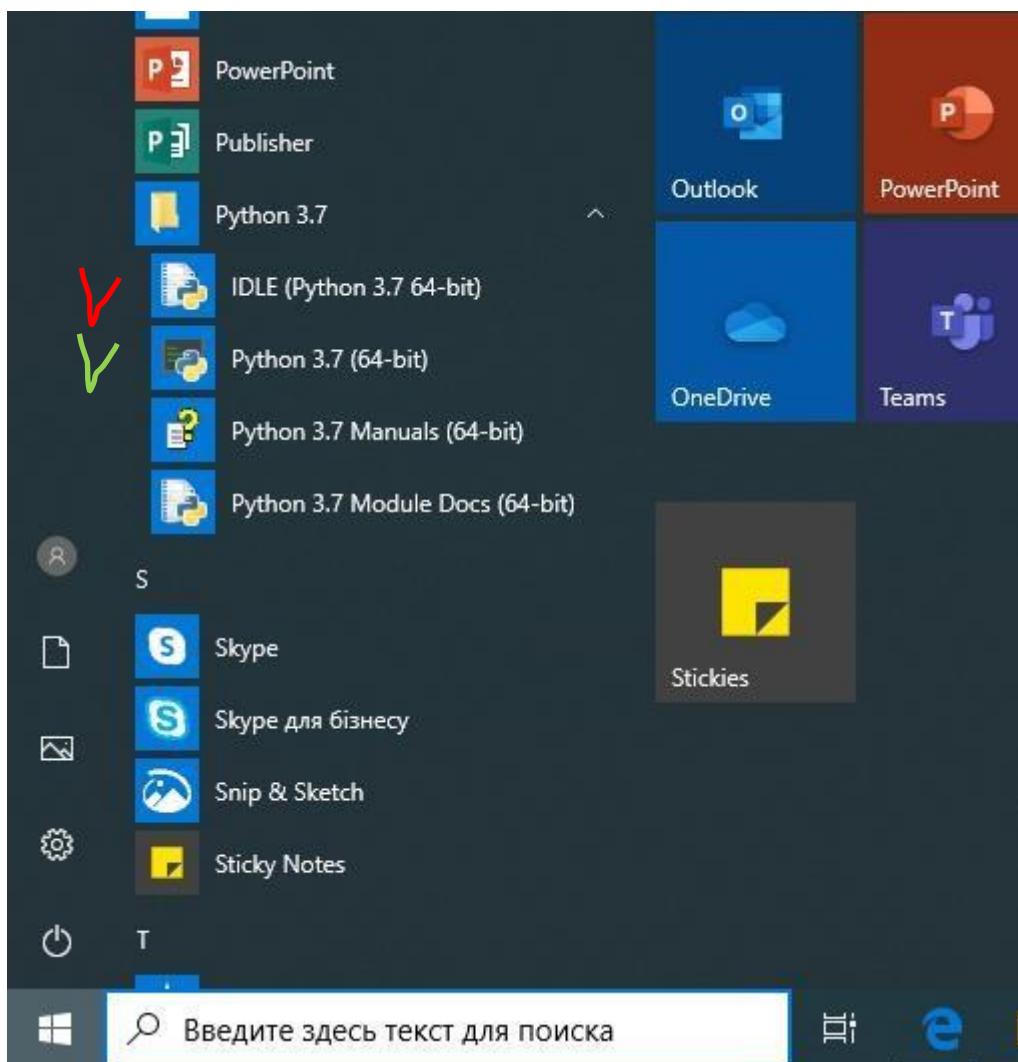
The screenshot shows a Windows desktop environment. A browser window is open to the Python Software Foundation's download page for Python 3.7.4. The 'Files' section lists various installation packages. Two specific items are highlighted with a red bracket: 'Windows x86-64 executable installer' and 'Windows x86 web-based installer'. Both of these items have a red border around them.

Інсталятор можна використовувати віддалений або скачати і запускати локально. Рекомендуємо у вікні інсталятору установити «галочку» “Add Python 3.7 to PATH”. Таким чином операційна система автоматично сформує шлях до іnstальованого інтерпретатору *Python* і зберігатиме його, як значення відповідної системної сталої. Доцільність очевидна.



Оточення Python

У процесі інсталяції у меню «Start» (рис.) буде сформована папка *Python 3.7*, яка містить ярлички для запуску *Python* (червоний і зелений маркер), а також для звернення до докладного довідника із мови *Python* (білий маркер), який входить у комплект поставки.



Python може працювати в оточенні безпосередньо операційної системи і спеціальної оболонки – середовище програміста.

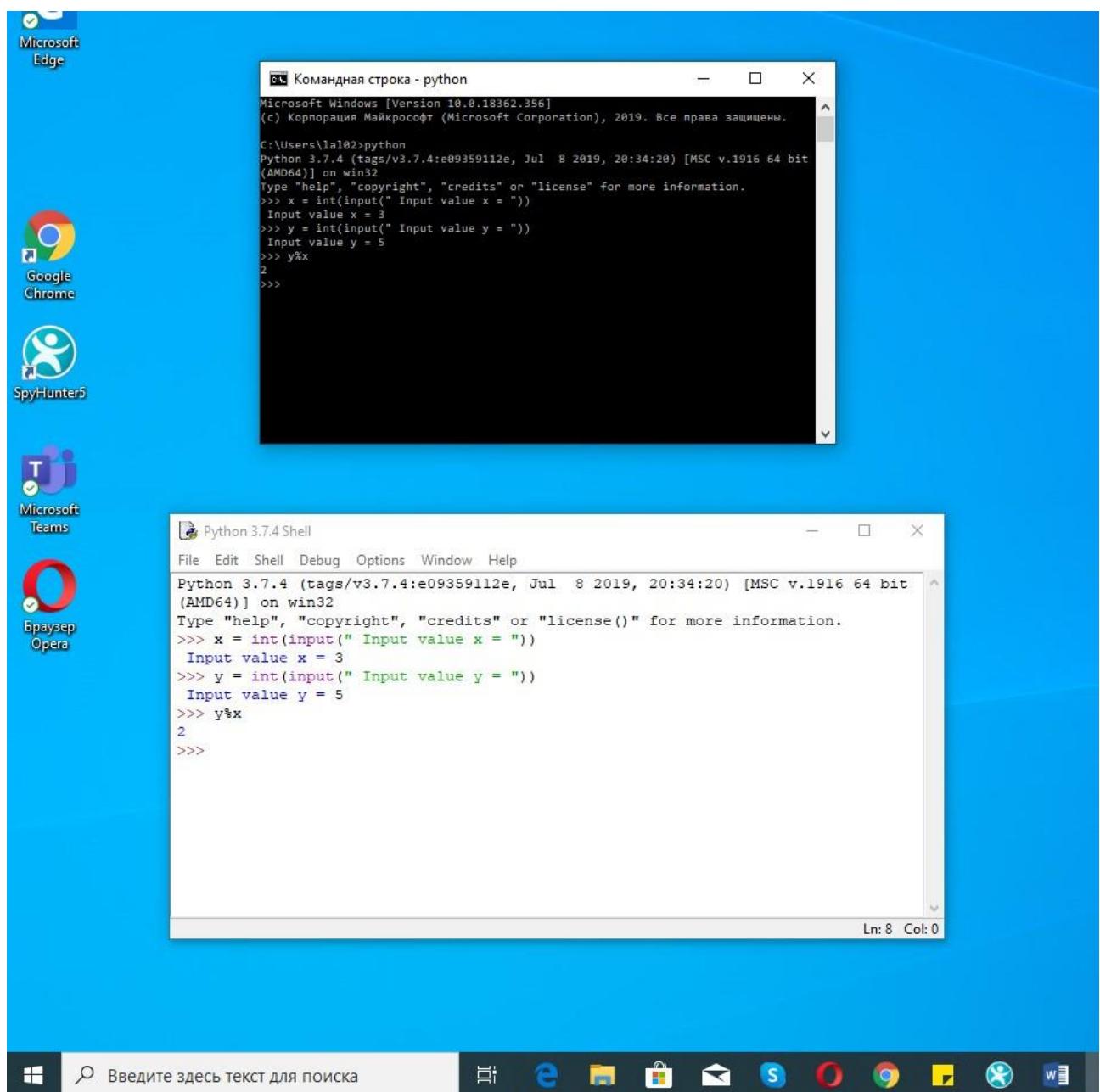
Клик на іконці «Python 3.7 (64 bit)» (зелений маркер) або «IDLE (Python 3.7 64 bit)» (червоний маркер) безпосередньо запускає інтерпретатор мови *Python*. У першому випадку відкривається вікно консолі операційної системи із запрошенням ввести інструкцію інтерпретатору. Тобто, ядро *Python* працює безпосередньо у середовищі OS.

У другому випадку OS запускає оболонку IDLE і робота з *Python* відбувається у в оточенні, яке надає програмісту набагато більше комфорту і

можливостей, ніж безпосередньо OS. Наприклад, більш просунутий редактор текстів та багато ін.

Разом з цим, оболонки накладають і певні обмеження. Наприклад, IDLE досить жорстко обмежує графічні можливості Python. Одна з найбагатшим інструментарієм бібліотека colorama не підтримується цією оболонкою.

В обох випадках після натиснення клавіши Enter інструкція виконується, а безпосередньо під нею виводиться результат, якщо це не заперечено. Після виконання інструкції з'являється запрошення на введення чергової інструкції і т.д.



IDLE – далеко не єдине середовище Python-програміста. У якості альтернатив можна вказати посилання на такі оболонки:

- АТОМ ([Atom](#));
- Geany ([Home | Geany](#))
- PyCharm ([Download PyCharm: Python IDE for Professional Developers by JetBrains](#)). Цілком придатним є варіант інсталяції **Community**.

Варто відмітити, що оболонка IDLE виглядає скромною порівняно з іншими. Проте, їх переваги набувають значення тільки у професійній діяльності і надмірними для начальних цілей, що ставимо перед собою. Простота використання і повнота інструментарію є достатньою підставою зупинитися саме на оболонці IDLE.

Інсталяція менеджера PIP й оболонки IDLE

В залежності від OS, ці кроки виконуються по різному,

Windows 10. Менеджер пакетів pip

Python – це досить компактне ядро, яке містить інтерпретатор мови програмування надвисокого рівня (VHLL), програмне оточення ядра (оболонку) і величезну кількість проблемно орієнтованих пакетів, більшість з яких написано мовою **Python**. Одна із переваг полягає у тому, що все багатство повністю безкоштовне і може бути вільно скачане. Ресурси постійно оновлюються. Більш того вони є відкритими і у тому розумінні, що кожен бажаючий може приєднатися до спільноти і поповнити ресурс власними розробками.

Одним з таких ресурсів є [PyPI . The Python Package Index](#). Зв'язок вашої інсталяції **Python** з каталогом PyPI здійснюватиме менеджер **pip**, який треба встановити

Інсталяція

Менеджер **pip** зв'язує середовище Python з ресурсом [PyPI . The Python Package Index](#) каталогом [Python Package Index \(PyPI\)](#) і призначений для установки й управління проблемно орієнтованими пакетами.

Робота з менеджером здійснюється через командний рядок операційної системи. Python Release для Windows 10, як правило, містить менеджер **pip** і він інсталюється автоматично. Разом з цим, має сенс перевірити правильність установки, це по-перше. А по-друге, ознайомитися із переліком команд, якими менеджер управляється:

cmd pip --help (а можна і просто ***pip***).

Результат показаний на рисунку.

```
с4. Командная строка
Microsoft Windows [Version 10.0.18362.476]
(c) Корпорация Майкрософт (Microsoft Corporation), 2019. Все права защищены.

C:\Users\lal02>pip --help

Usage:
  pip <command> [options]

Commands:
  install                  Install packages.
  download                Download packages.
  uninstall               Uninstall packages.
  freeze                  Output installed packages in requirements format.
  list                    List installed packages.
  show                    Show information about installed packages.
  check                  Verify installed packages have compatible dependencies.
  config                  Manage local and global configuration.
  search                  Search PyPI for packages.
  wheel                  Build wheels from your requirements.
  hash                   Compute hashes of package archives.
  completion             A helper command used for command completion.
  debug                  Show information useful for debugging.
  help                   Show help for commands.

General Options:
  -h, --help              Show help.
  --isolated              Run pip in an isolated mode, ignoring environment variables and user configuration.
  -v, --verbose            Give more output. Option is additive, and can be used up to 3 times.
  -V, --version            Show version and exit.
  -q, --quiet              Give less output. Option is additive, and can be used up to 3 times (corresponding to WARNING, ERROR, and CRITICAL logging levels).
  --log <path>            Path to a verbose appending log.
  --proxy <proxy>          Specify a proxy in the form [user:passwd@]proxy.server:port.
  --retries <retries>      Maximum number of retries each connection should attempt (default 5 times).
  --timeout <sec>          Set the socket timeout (default 15 seconds).
  --exists-action <action> Default action when a path already exists: (s)witch, (i)gnore, (w)ipe, (b)ackup, (a)bort.
  --trusted-host <hostname> Mark this host as trusted, even though it does not have valid or any HTTPS.
  --cert <path>            Path to alternate CA bundle.
  --client-cert <path>     Path to SSL client certificate, a single file containing the private key and the certificate in PEM format.
  --cache-dir <dir>        Store the cache data in <dir>.
  --no-cache-dir           Disable the cache.
  --disable-pip-version-check
                           Don't periodically check PyPI to determine whether a new version of pip is available for download. Implied with --no-index.
  --no-color               Suppress colored output

C:\Users\lal02>
```

Управління пакетами за допомогою pip.

Пакети установлюються через командний рядок за допомогою інструкції

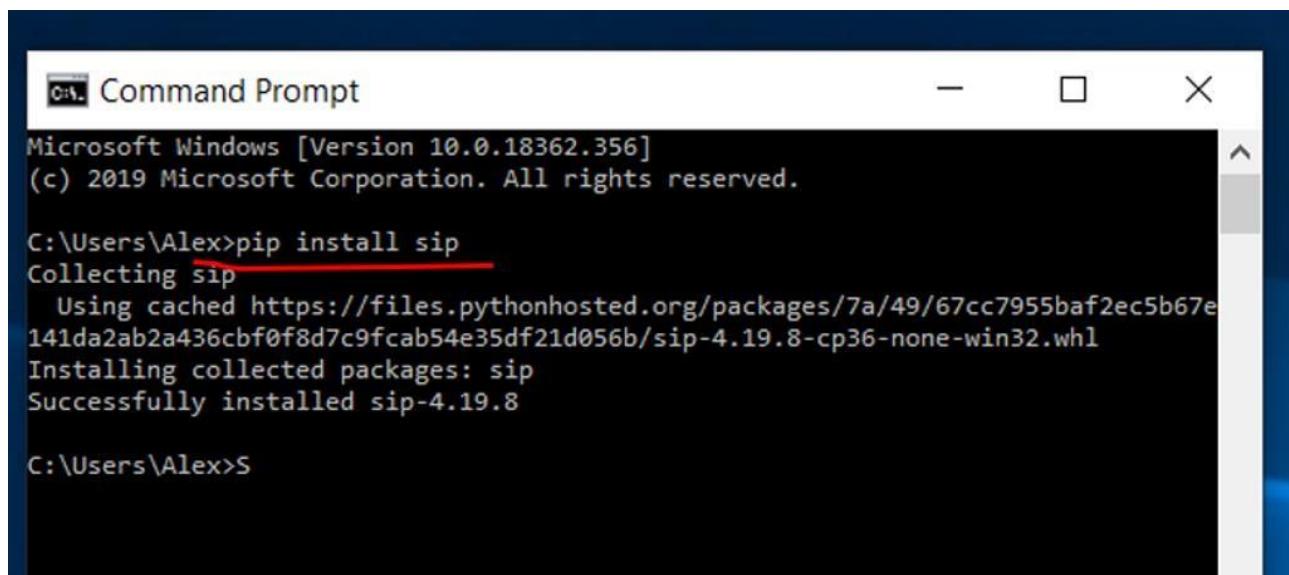
pip install [some-package-name].

Видалити пакет можна інструкцією

pip delete [some-package-name],

В цих інструкціях *some-package-name* – назва пакету.

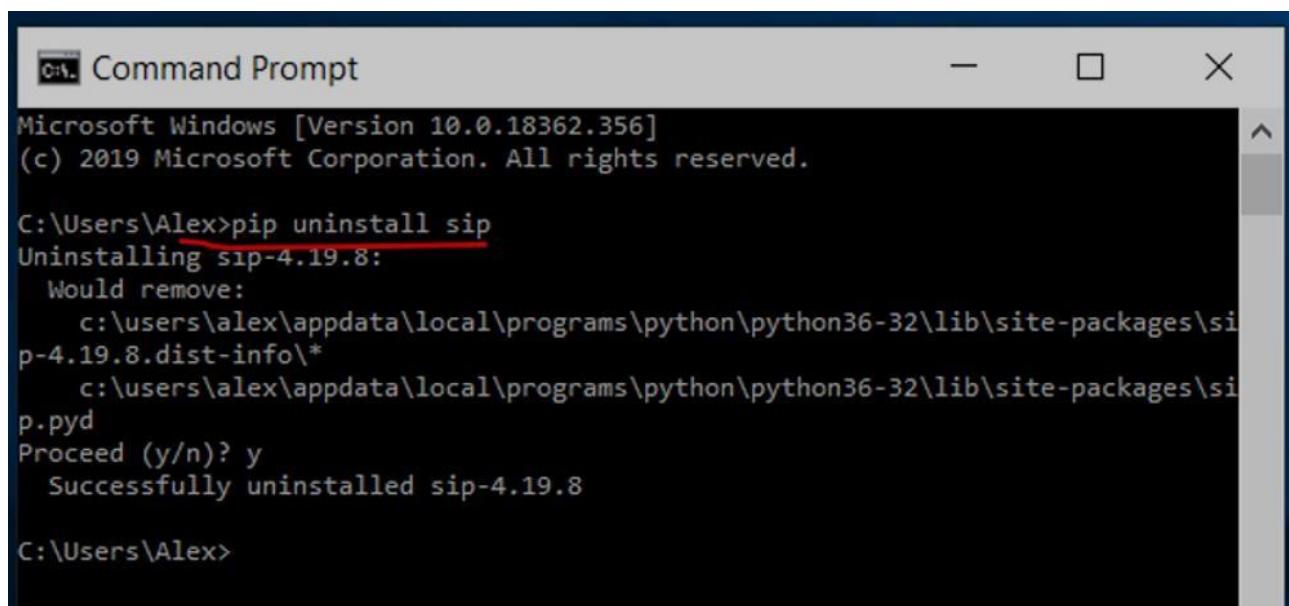
Приклад. Програмне забезпечення, написане на **Python** може активно використовувати і взаємодіяти з ПЗ і бібліотеками, написаними мовами С і С++. Пакет **sip** містить інструменти для розроблення програмних модулів, які забезпечуватимуть інтерфейс програм, написаних на Python, і на С й С++. Пакет розроблений Філом Томпсоном (<http://riverbankcomputing.co.uk/software/sip/intro>), зберігається у каталозі **PyPI** й активно підтримується. При необхідності він може бути встановлений на комп’ютер, а потім видалений, що показано на рисунку червоним маркером.



```
Command Prompt
Microsoft Windows [Version 10.0.18362.356]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Alex>pip install sip
Collecting sip
  Using cached https://files.pythonhosted.org/packages/7a/49/67cc7955baf2ec5b67e141da2ab2a436cbf0f8d7c9fcab54e35df21d056b/sip-4.19.8-cp36-none-win32.whl
Installing collected packages: sip
Successfully installed sip-4.19.8

C:\Users\Alex>
```



```
Command Prompt
Microsoft Windows [Version 10.0.18362.356]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Alex>pip uninstall sip
Uninstalling sip-4.19.8:
Would remove:
  c:\users\alex\appdata\local\programs\python\python36-32\lib\site-packages\sip-4.19.8.dist-info\*
  c:\users\alex\appdata\local\programs\python\python36-32\lib\site-packages\sip.pyd
Proceed (y/n)? y
Successfully uninstalled sip-4.19.8

C:\Users\Alex>
```

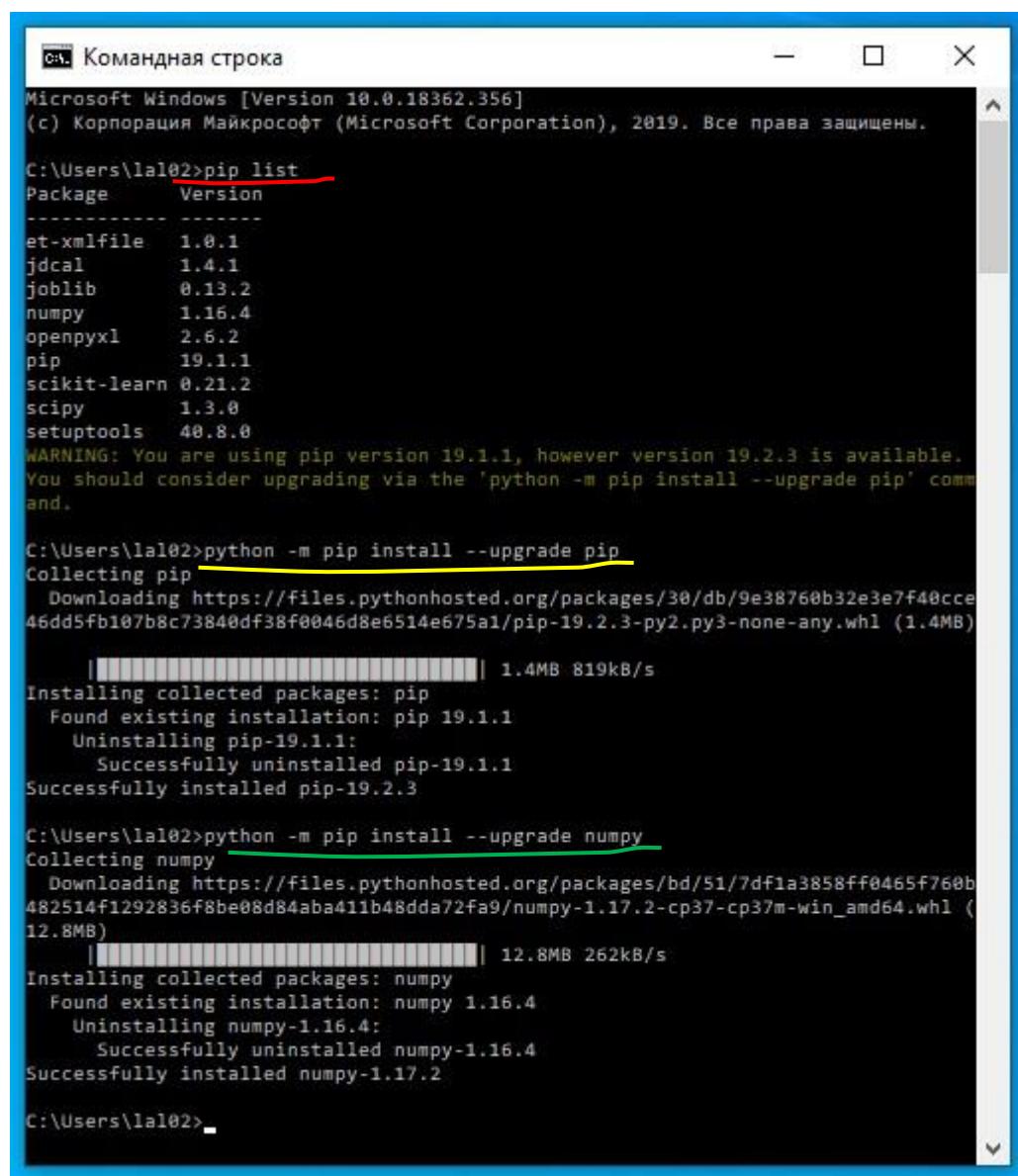
Оновлення пакетів за допомогою pip.

Ресурс *PyPI* постійно оновлюється й удосконалюється. Після тривалого використання доцільно перевірити їй, при необхідності, оновити пакети до актуальної версії.

Список пакетів, установлених у даний момент на комп'ютері, та їх версії можна вивести на екран за допомогою інструкції, показаною на рисунку червоним маркером.

У цьому списку, природно, буде і сам менеджер *pip* та його версія.

Оновлення пакетів здійснюється інструкцією, показаною на рисунку жовтим маркером.



The screenshot shows a Windows Command Prompt window titled "Командная строка". The command `pip list` is run, displaying a table of installed packages and their versions. The command `python -m pip install --upgrade pip` is run, followed by the upgrade process for the pip package. Finally, the command `python -m pip install --upgrade numpy` is run, followed by the upgrade process for the numpy package.

```
C:\Users\lal02>pip list
Package    Version
-----
et-xmlfile  1.0.1
jdcal      1.4.1
joblib     0.13.2
numpy       1.16.4
openpyxl    2.6.2
pip         19.1.1
scikit-learn 0.21.2
scipy       1.3.0
setuptools  40.8.0
WARNING: You are using pip version 19.1.1, however version 19.2.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

C:\Users\lal02>python -m pip install --upgrade pip
Collecting pip
  Downloading https://files.pythonhosted.org/packages/30/db/9e38760b32e3e7f40cce46dd5fb107b8c73840df38f0046d8e6514e675a1/pip-19.2.3-py2.py3-none-any.whl (1.4MB)

    |████████████████████████████████| 1.4MB 819kB/s
Installing collected packages: pip
  Found existing installation: pip 19.1.1
  Uninstalling pip-19.1.1:
    Successfully uninstalled pip-19.1.1
Successfully installed pip-19.2.3

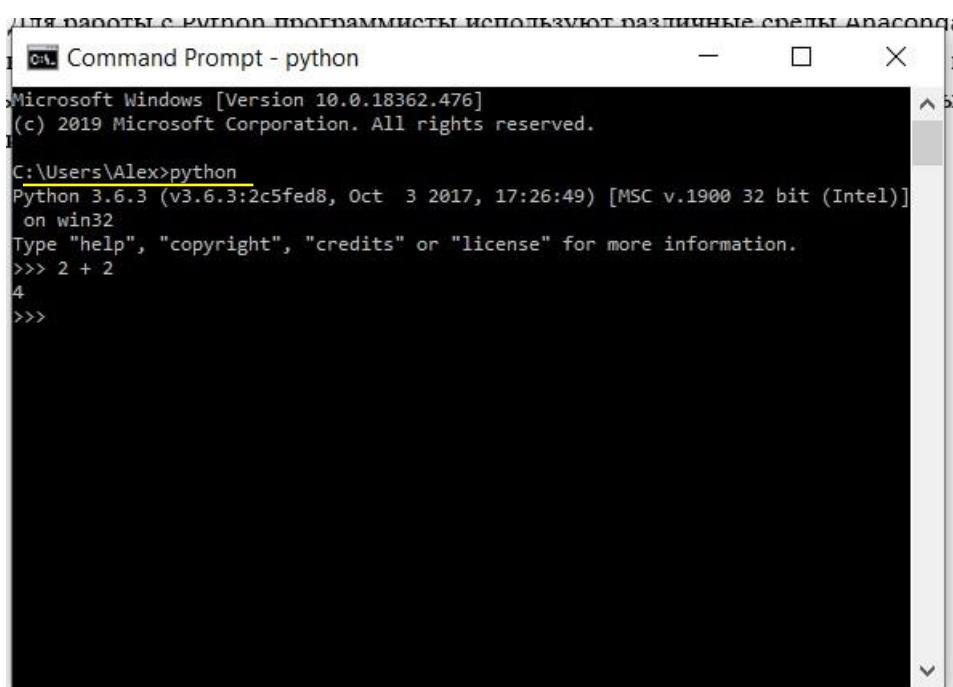
C:\Users\lal02>python -m pip install --upgrade numpy
Collecting numpy
  Downloading https://files.pythonhosted.org/packages/bd/51/7df1a3858ff0465f760b482514f1292836f8be08d84aba411b48dda72fa9/numpy-1.17.2-cp37-cp37m-win_amd64.whl (12.8MB)
    |████████████████████████████████| 12.8MB 262kB/s
Installing collected packages: numpy
  Found existing installation: numpy 1.16.4
  Uninstalling numpy-1.16.4:
    Successfully uninstalled numpy-1.16.4
Successfully installed numpy-1.17.2

C:\Users\lal02>
```

На цьому рисунку показаний і лістинг дій OS Windows 10 за цими інструкціями при оновленні самого менеджера *pip* (жовтий маркер) з версії 19.1.1 до актуальної версії 19.2.3, а також пакету *numpy* (зелений маркер) з версії 1.16.4 до актуальної версії 1.17.2.

OS Windows 10. Оболонка IDLE

Працювати з інтерпретатором *PyPy* можна безпосередньо, через командний рядок OS. Кожен бажаючий може це спробувати й упевнитися, на скільки це не комфортно.



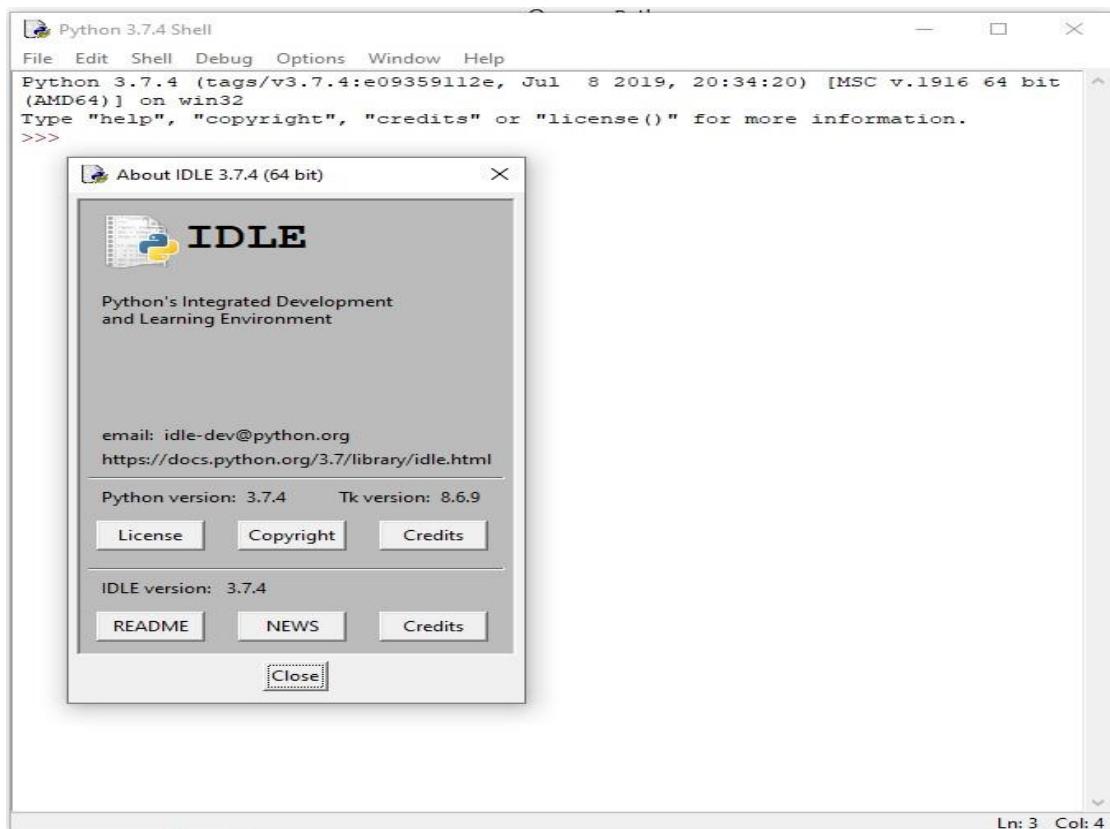
```
Microsoft Windows [Version 10.0.18362.476]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Alex>python
Python 3.6.3 (v3.6.3:2c5fed8, Oct  3 2017, 17:26:49) [MSC v.1900 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 2
4
>>>
```

Для розроблення і виконання програм, взаємодії із операційною системою або іншими програмами, з Інтернет, БД і т.і., можна використовувати різні середовища – PyCharm, IDLE, Anaconda, MS Visual Studio та ін., або власний інтерфейс, написаний на Python.

У даному посібнику використовується просте, ї, водночас, професійне середовище IDLE.

Python Release для Windows 10, повторюємо, містить IDLE і оболонка інсталюється автоматично. На робочому столі й у меню Start з'являються відповідні ярлики для запуску оболонки. Робоче вікно оболонки має вигляд (рис.). На цьому ж рисунку наведені дані про оболонку, що використовується.



OS Ubuntu. Оболонка IDLE

Python Release для Linux, Ubuntu є інші не містить середовище IDLE й менеджер pip. Їх треба встановлювати окремо і при цьому треба враховувати особливості цих OS.

Python 3.6 входить у Release Ubuntu 18.04 LTS і вище, і, таким чином, повинен встановлюватися за замовчанням. Упевнитися можна інструкцією:

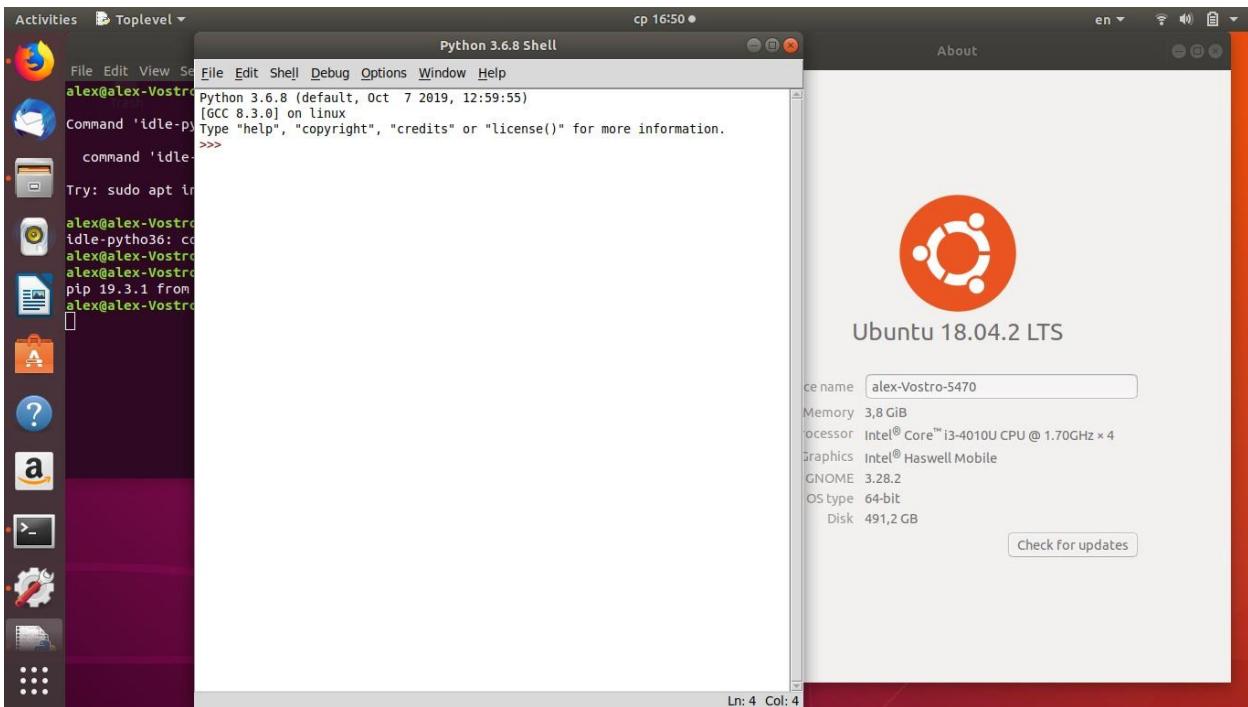
***Ctrl+Alt+T sudo
python3.6***

Результат мало чим відрізняється від наведеного на рис для OS Windows 10.
Входим у середовище OS Ubuntu інструкцією

exit()

Оболонка *idle* запускається інструкцією (рисунок)

idle-python3.6



OS Ubuntu. Менеджер пакетів pip

Менеджер пакетів pip можна інсталювати так:

```
 wget https://bootstrap.pypa.io/get.pip.py sudo python3.6 -m get-pip.py
```

Взнати версію встановленого менеджера можна інструкцією

```
 pip -version
```

Всі проблемно орієнтовані пакети встановлюються за допомогою інструкції

```
 sudo python3.6 -m pip install [package name]
```

Оновити менеджер й пакети можна

```
 pip install --upgrade pip pip install --upgrade [package name]
```

Завдання:

1. Встановити Python на свій комп'ютер у відповідності із наявною операційною системою.
2. Встановити оболонку IDLE. Запустити і щоб упевнитися у працездатності, виконати декілька простих обчислень у режимі калькулятора.
3. Упевнитися, що менеджер pip встановлений. Якщо ні, то установити його, використовуючи надані рекомендації.
4. З'ясувати, які пакети встановлені за замовчанням. Оновити будь-який з них до актуальної версії і таке інше. Тобто, виконати перелік команд: **pip help** – довідка із досяжних команд. **pip install package_name** - установка пакета(ів). **pip uninstall package_name** – видалення пакета(ів). **pip list** - список установлених пакетів.

pip show package_name – показує інформацію про встановлений пакет. **pip search** - пошук пакету за іменем.

pip --proxy user:passwd@proxy.server:port – використання з проксі сервером. **pip install -U** - обновлення пакету(ів). **pip install --force-reinstall** - при оновленні, переустановити пакет, навіть якщо він останньої версії.

Результати виконання кожного з пунктів проілюструвати скріншотами.

Лабораторна робота № 3 **Тема:**

«Python – мова скриптів»

Мета:

1. Дати поняття скрипту, як основної семантичної одиниці мови надвисокого рівня Python.
2. На прикладах проілюструвати універсальність сучасної мови Python.

Теоретичний мінімум

Поняття скрипта (<https://whatis.techtarget.com/definition/script>)

Python часто називають мовою скриптів, або сценаріїв. Слово «сценарій» є одним із перекладів слова “script”.

Скрипт – це семантична одиниця (атомарний вираз) мови надвисокого рівня.

Скрипт для програміста є певним виразом, часто зовсім невеликим. Але при інтерпретації внутрішнє, машинне, поданні скрипту – це послідовність інструкцій, або, навіть, ціла програма, що виконуватиметься інтерпретатором.

У цьому полягає кардинальна відміна від мови високого рівня. У мові високого рівня (C++, C# та ін) семантичною одиницею (атомарний вираз мови) є **оператор**. Після компіляції оператор, у внутрішньому поданні мови, є інструкцією безпосередньо процесору комп’ютера.

Завдяки такій архітектурі Python-програміст мислить блоками сумісних дій, код коротший і, відповідно, менше прикрих помилок, писати правильні програми набагато легше.

Далі наведений приклад коду, записаний мовами Python і C++. Дії полягають у введені масивів, додаванні, множенні, ділені відповідних елементів масивів і виведені результату. Кожний може сам зробити висновок, чи справедливі слова попереднього абзацу.

```
import numpy as np
x, y = np.array([1,2, 3]), np.array( [4, 5, 6])
print('x + y= ', x+y, ' x * y= ', x*y, ' x / y= ', x/y )
x + y= [5 7 9]   x * y= [ 4 10 18]   x / y= [0.25 0.4 0.5 ]
>>>
```

```

9 #include <iostream>
10
11 using namespace std;
12
13 int main()
14 {
15     float x[3] = {1, 2, 3};
16     float y[3] = {4, 5, 6};
17
18     int lenx = sizeof(x)/sizeof( *x);
19     cout<<"x + y = [ ";
20
21     for (int i=0; i < lenx; i++)
22     {
23         cout<<x[i]+y[i]<<" ";
24     }
25     cout<<" ]";
26
27     cout<<"\n x * y = [ ";
28     for (int i=0; i < lenx; i++)
29     {
30         cout<<x[i]*y[i]<<" ";
31     }
32     cout<<" ]";
33
34     cout<<"\n x / y = [ ";
35     for (int i=0; i < lenx; i++)
36     {
37         cout<<x[i]/y[i]<<" ";
38     }
39     cout<<" ]";
40
41     return 0;
42 }
43

```

```
x + y = [ 5 7 9 ]    x * y = [ 4 10 18 ]    x / y = [ 0.25 0.4 0.5 ]
```

Програма на С – це послідовність операторів. На відзнаку від цього інтерпретатор Python, при виконанні, наприклад, скрипта print самостійно виконує цілу низьку дій: аналізує вираз скрипта, визначає тип даних, формує цикли для їх оброблення і т.д.

Природно, що програми, написані мовами VHLL інтерпретуються і виконують суттєво повільніше, ніж скомпільовані. Разом з цим, час розроблення й впровадження такого ПЗ досить часто набагато менший і у процесі експлуатації суттєвих зауважень до часових характеристик також не виникає. **Чому, ваша думка?** Доцільно пригадати поняття «життєвий цикл програми».

Скрипти – високий рівень універсальності мови. У лабораторній роботі це ілюструє, далеко не повний, ряд прикладів із різноманітних предметних галузей.

Універсальність Python. Приклади діалогу з інтерпретатором PyPy Приклад

1. Взаємодія із операційною системою.

Пакет **OS** містить модулі й функції, які дають змогу писати скрипти для взаємодії із операційною і файловою системами комп’ютера.

На рисунку показаний сценарій діалогу з OS Windows 10 із формуванням шляху доступу до документації Python, встановленого на комп’ютері:

1. Завантаження пакету **OS**, який містить увесь необхідний інструментарій.

2. Функція **getcwd()** пакету **OS**:

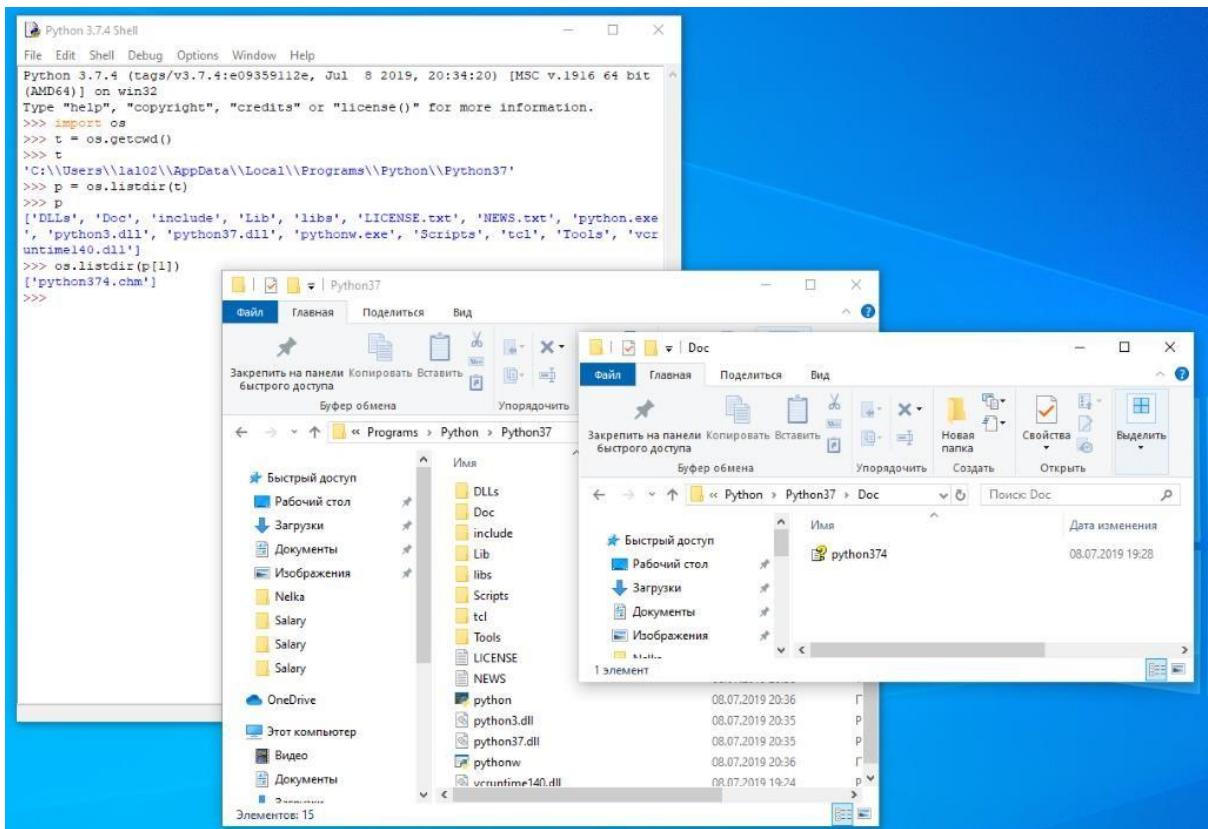
- знаходить на жорсткому диску шлях до робочого каталогу Python;
- створює об’єкт у вигляді рядка, який описує це шлях;
- створює змінну *t*, значенням якої є адреса (посилання) цього рядка.

3. Функція **listdir(t)** створює список об’єктів, які розташовані у робочому каталогі, а також змінну *p*, яка містить адресу цього списку.

4. Оператор візуально визначає, що папка із документацією є другою у цьому списку.

5. Тепер функція **listdir(p[1])** застосовується до другого елементу списку (нумерація у **Python**, як і у C, починається з нуля). Вміст папки виводиться на консоль. Папка містить тільки один об’єкт. Це файл із розширенням імені *.chm, що, за замовчанням, відповідає файлу документації

На рисунку показане вікно оболонки IDLE зі скриптами та результатами діалогу, а також, для ілюстрації, відповідні вікна, створені Windows 10.



При інсталяції операційна система запам'ятує шлях до інтерпретатора Python. Відповідно, всі результати, отримані при роботі програм, будуть записуватися за замовчанням у директорію, де зберігається інтерпретатор, інші службові файли та папки. Це, по-перше, не поганий стиль, а по-друге, може мати катастрофічні наслідки (чому?). Отже, доцільно створити окрему робочу директорію і вказати системі шлях до неї.

6. Скрипт організації доступу до своєї робочої директорії

```
>>> import os
>>> z = os.getcwd()
>>> print(z)
C:\Users\la102\AppData\Local\Programs\Python\Python38
>>> new_dir= z.replace( 'Python38', 'WORK')
>>> os.chdir(new_dir)
>>> os.getcwd()
'C:\\\\Users\\\\la102\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\WORK'
```

Python має добре розвинutий інструментарій для створення скриптів із цілим спектром програмним пакетів й додатків. Наступний приклад ілюструє можливість взаємодії з пакетом Microsoft Office, а саме, з табличним процесором Excel. [Приклад 2. Доступ до комірки таблиці, створеної пакетом Excel.](#)

Офісний пакет Excel надає величезні спроможності щодо інтерактивного оброблення даних, поданих таблицями. Проте, продуктивність роботи стає неприпустимо низькою при оброблені великих за розмірами таблиць. Спроможності щодо Excel автоматизації процесів оброблення невеликі.

На *Python* можна писати сценарії для автоматизації практично всіх етапів роботи з такими таблицями. Для цього використовуються спеціалізовані пакети *pandas*, *openpyxl*, *xlrd*, *xlutils*, *pyexcel* та ін. (написані, доречі, на самому *Python*). У даному посібнику використовується пакет *openpyxl*.

Припускається, що вихідна таблиця зберігається у файлі “Книга1.xlsx”, який, у свою чергу, зберігається у папці ‘WORK’, яка розміщена у папці “Python3632”. На рисунку показана ця Excel-таблиця.

Далі наведений сценарій (з коментарями), що забезпечує доступ до таблиці і вмісту комірок таблиці, з чого, звичайно, починається оброблення табличних даних. Мета – у переліку продуктів замінити назву продукту “Apples” на “potato”. Код виконується у режимі діалогу виключно для демонстрації можливостей *Python*. В реальності він цілком придатний для розроблення автоматичної програми.

Цей текст є копією лістингу діалогу. Кольори відповідають кольорам тексту в оболонці IDLE..

The screenshot shows a Microsoft Excel window titled "Книга1 - Excel". The ribbon menu is visible at the top, with "Основы" (Основи) selected. The formula bar shows the cell reference "B2" and the value "Apples". The main area displays a table with three columns: Column1, Column2, and Column3. The data rows are as follows:

	Column1	Column2	Column3
2	04.05.2014 13:34	Apples	73
3	04.05.2014 3:41	Cherries	85
4	04.06.2014 12:46	Pears	14
5	04.08.2014 8:59	Oranges	52
6	04.10.2014 2:07	Apples	152
7	04.10.2014 18:10	Bananas	23
8	04.10.2014 2:40	Strawberries	98
9			

Лістинг:

```
>>> # Завантажуємо необхідні пакети OS – робота з операційною системою,  
>>> # openpyxl – для роботи з електронними  
таблицями  
>>> # Для зручності довге ім'я пакету openpyxl змінено на коротке локальне  
ім'я ex  
  
>>> import os, openpyxl as ex  
  
>>> # Організація доступу до файлу 'Книга1.xlsx':  
>>> # 1. Визначаємо актуальну робочу директорію Python:  
>>> way = os.getcwd()  
>>> way  
'C:\\\\Users\\\\Alex\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32'  
  
>>> # 2. Використовуємо метод 'replace' й формуємо строкову шлях до  
файлу  
'Книга1.xlsx':  
>>> new_way = way.replace( 'Python36-32' , 'WORK' )  
>>> new_way  
'C:\\\\Users\\\\Alex\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\WORK'  
  
>>> # 3. Встановлюємо нову робочу директорію - доступ до файлу  
'Книга1.xlsx':  
>>> os.chdir(new_way)  
>>> os.getcwd()  
'C:\\\\Users\\\\Alex\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\WORK'  
  
>>> # 4. 'Книга1.xlsx' зберігається у папці 'WORK' (рис. ). Метод  
'load_workbook' пакету 'openpyxl' читує файл і формує у RAM об'єкт, на який  
вказуватимемо змінна wb.  
>>> wb = ex.load_workbook('Книга1.xlsx')  
  
>>> # 5. За допомогою методу 'get_sheet_names' пакету 'openpyxl' отримуємо  
список аркушів Книги1 (рис. )  
>>> wb.get_sheet_names()  
DeprecationWarning: Call to deprecated function get_sheet_names (Use  
wb.sheetnames). ['Аркуш2', 'Аркуш1']  
>>> # 6. Створюємо об'єкт, що відповідає аркушу 'Аркуш2'
```

```
>>> sheet = wb.get_sheet_by_name('Аркуш2')
```

DeprecationWarning: Call to deprecated function get_sheet_names (Use
wb[sheetnames]).

>>> # 7. Нас цікавить комірка B2, яка містить значення 'Apple' (рис.).

Вказуємо методу cell її атрибути й отримуємо доступ до її значення

```
>>> sheet.cell( row = 2, column = 2 )
```

```
>>> sheet.cell( row = 2, column = 2 ).value
```

'Apple'

>>> # 8. Зміняємо значення комірки B2 'Apple' на 'potato'

```
>>> sheet.cell( row = 2, column = 2, value = 'potato' )
```

>>> # 9. Змінений об'єкт wb конвертуємо у формат таблиці Excell й
поміщаємо у новий файл 'Книга2.xlsx', який створюється автоматично

```
>>> wb.save('Книга2.xlsx')
```

Кінцевий результат виконання коду показаний на рис.

```

>>> a = [[i, sheet.cell(row= i+1, column= 2).value] for i in range(1, 134) ]
>>> a
[[1, 24012], [2, 24340], [3, 24823], [4, 25411], [5, 25964], [6, 26514], [7, 269
99], [8, 27462], [9, 27856], [10, 28381], [11, 29070], [12, 29753], [13, 30506],
[14, 31154], [15, 31810], [16, 32476], [17, 33234], [18, 34063], [19, 34984], [
20, 35825], [21, 36560], [22, 37241], [23, 38074], [24, 39014], [25, 40008], [26
, 41109], [27, 42065], [28, 42982], [29, 43628], [30, 44334], [31, 44998], [32,
45887], [33, 46763], [34, 47677], [35, 48500], [36, 49043], [37, 49607], [38, 50
414], [39, 51224], [40, 52043], [41, 52843], [42, 53521], [43, 54133], [44, 5477
1], [45, 55607], [46, 56455], [47, 57264], [48, 58111], [49, 58842], [50, 59493]
, [51, 60166], [52, 60995], [53, 61851], [54, 62823], [55, 63929], [56, 64849],
[57, 65656], [58, 66575], [59, 67597], [60, 68794], [61, 69884], [62, 71056], [6
3, 72168], [64, 73158], [65, 74219], [66, 75490], [67, 76808], [68, 78261], [69,
79750], [70, 80949], [71, 81957], [72, 83115], [73, 84548], [74, 86140], [75, 8
7872], [76, 89719], [77, 91356], [78, 92820], [79, 94436], [80, 96403], [81, 985
37], [82, 100643], [83, 102971], [84, 104958], [85, 106757], [86, 108415], [87,
110085], [88, 112059], [89, 114497], [90, 116978], [91, 119074], [92, 121215], [
93, 123303], [94, 125798], [95, 128228], [96, 130951], [97, 133787], [98, 135894
], [99, 138068], [100, 140479], [101, 143030], [102, 145612], [103, 148756], [10
4, 151859], [105, 154335], [106, 156797], [107, 159702], [108, 162660], [109, 16
6244], [110, 169472], [111, 172712], [112, 175678], [113, 178353], [114, 181237]
, [115, 184734], [116, 188106], [117, 191671], [118, 195504], [119, 198634], [12
0, 201305], [121, 204932], [122, 208959], [123, 213028], [124, 217661], [125, 22
2322], [126, 226462], [127, 230236], [128, 234584], [129, 239337], [130, 244734]
, [131, 250538], [132, 256266], [133, 261034]]

```

	A	B	C	D	E	F
1	Column1	Column2	Column3			
2	04.05.2014 13:34	potato	73			
3	04.05.2014 3:41	Cherries	85			
4	04.06.2014 12:46	Pears	14			
5	04.08.2014 8:59	Oranges	52			
6	04.10.2014 2:07	Apples	152			
7	04.10.2014 18:10	Bananas	23			
8	04.10.2014 2:40	Strawberries	98			
9						

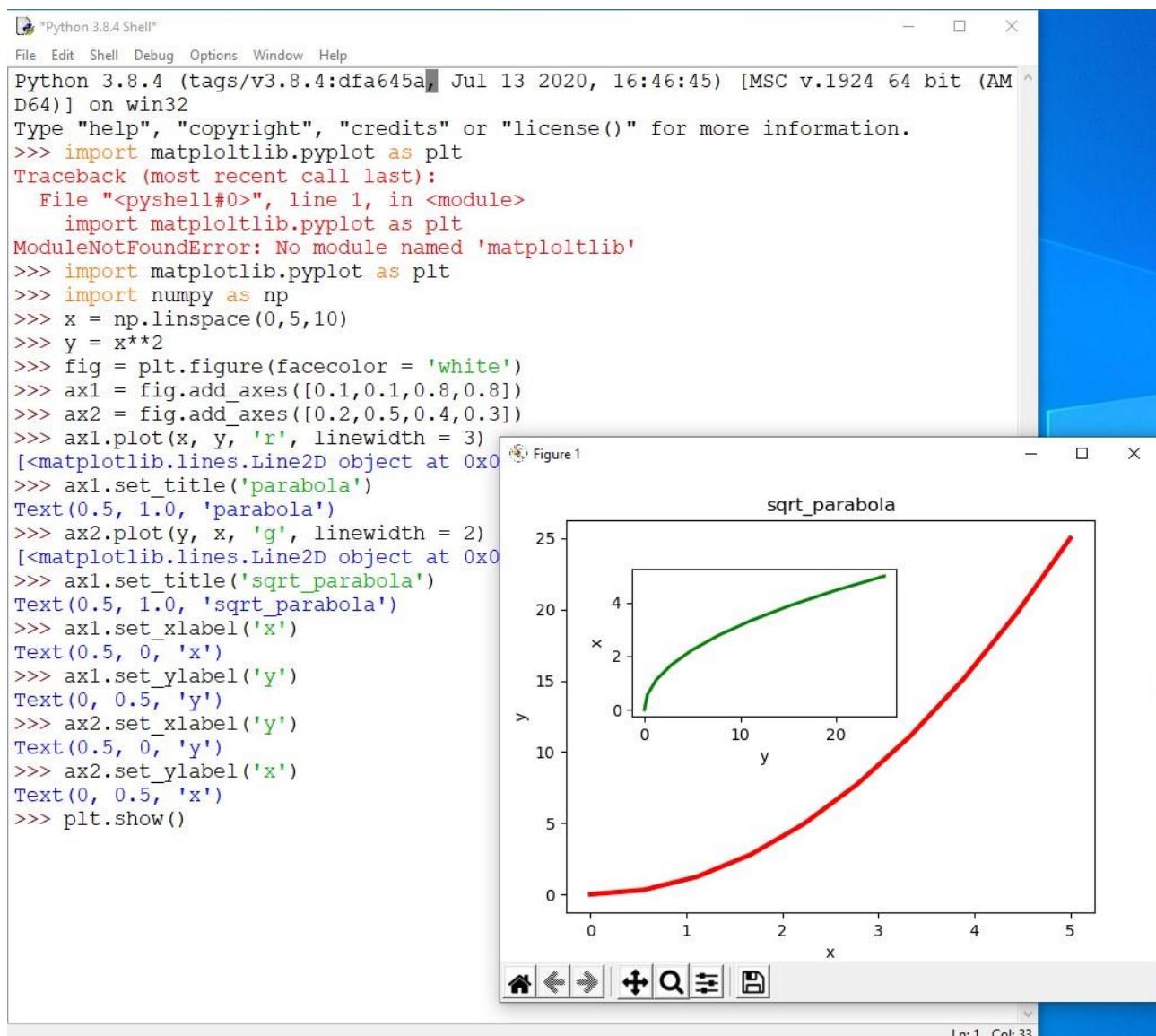
Зауважимо, що наведений приклад, крім коментарів, містить повідомлення операційної системи типу

DeprecationWarning: Call to deprecated function get_sheet_names
(Use wb.sheetnames).

Це зроблено навмисно, оскільки ілюструє інтелект інтерпретатора Python. Інтерпретатор встановив, що скрипт містить посилання на функцію версії старішої за ту, що встановлена на комп'ютері. Більш того, нова версія має інший формат, ніж стара! Не дивлячись на це, скрипт виконується, оскільки це не помилка. А саме – інтерпретатор викликає нову версію, виконує її і, навіть, підказує програмісту, як виглядає посилання на нову версію.

Приклад 3. Деякі графічні функції

Сучасний Python дає можливості програмувати графіку самого різного вигляду та призначення. Для цього розроблений цілий ряд бібліотек. Наступний простий приклад ілюструє роботу з однією із них (<https://matplotlib.org/3.5.1/index.html>).



Вище наведений код і рисунок. **Python** створює графічне вікно, у якому, методом **add_axes**, створюються два графічних об'єкти **ax1** і **ax2**. Розташування об'єктів визначається координатами відносно меж вікна. Програмісту надається

широкий асортимент методів, частина з яких використана у прикладі. Об'єкти є графіками квадратичної функції і оберненої до неї.

Python надає широкі можливості з оброблення інформації, закодованої не тільки у чисельних, але й символічних даних, зокрема, алгебраїчних. Такі спроможності надає, наприклад, модуль **SymPy**. *Приклад 4. Символьні перетворення*

```
>>> from sympy import *
>>> x = symbols('x')
>>> factor(x**4 + 5*x**3 - 7*x**2 - 5*x + 6)
(x - 1)**2*(x + 1)*(x + 6)
```

Вміє Python розв'язувати і рівняння

```
>>> solve((x-3)/(x-5) - x + 1)
[7/2 - sqrt(17)/2, sqrt(17)/2 + 7/2]
```

Завдання:

1. Взаємодія із операційною системою.
 - 1.1. Побудувати шлях доступу до файлу з документацією на своєму комп'ютері.
 - 1.2. Створити на своєму комп'ютері робочу директорію для Python, організувати шлях до неї.
2. Оброблення таблиць. У робочій директорії (завдання 1) створити excel-файл з таблицею. Замінити вміст деякої комірки.
3. Графічні об'єкти.
 - 3.1. Ввести код і подувати графічні об'єкти із прикладу. Встановити зміст параметрів використаних методів.
 - 3.2. Побудувати власний рисунок, аналогічний прикладу.
4. Символьні перетворення.
 - 4.1. Зі шкільного підручника взяти приклад з розкладання многочлена на множники і виконати його.
 - 4.2. Перевірити, чи вміє Python розв'язувати рівняння з комплексними коренями.

Що далі?

Основою програмування є поняття «базового типу даних». Цьому присвячена наступна робота.

Python настільки універсальна мова, що дає змогу писати програми на основі усіх відомих парадигм: структурне, об'єктно-орієнтоване, логічне і функціональне.

Декілька наступних лабораторних робот присвячені структурам даних Python і структурному програмуванню.

Діалог з Python через командний рядок на практиці широко використовується, але для управління певними програмними процесами, а не розроблення ПЗ.

Структурне програмування передбачає розроблення програм певної структури, а не діалог. Для розроблення програм (сценаріїв) використовується режим програмування. Для цієї мети розроблений цілий ряд середовищ програміста, про що написано вище.

Наступні лабораторні роботи передбачають використання саме такого режиму. Враховуючи навчальні цілі цього матеріалу, далі продовжуємо використовувати середовище IDLE, яке є простим, крос платформним і достатньо комфортним.

Лабораторна робота № 4

Тема: «Алгоритмічна мова Python. Режим калькулятора - лінійні програми» *Мета:*

1. *Отримати навички писати скрипти, які є послідовністю певних інструкцій інтерпретатору (режим калькулятора).*
2. *Познайомитися з бібліотеками для створення та оброблення структур даних *пітру* математичних функцій *math*, графічних функцій *matplotlib*, наукових розрахунків *scipy**

Теоретичний мінімум

Режим діалогу з інтерпретатором

Python – мова надвисокого рівня і його архітектура принципово відрізняється від мов високого рівня C++ та ін. Тут наведені тільки необхідні деталі. Докладно це питання розглядається, наприклад, у книжці Матта Харісона [2].

При знайомстві із **Python**, природно перші кроки робити у режимі діалогу (калькулятора), вводячи інструкції й одразу отримуючи результат. Канонічний приклад:

```
>>> print('Hello, World!') #Enter  
Hello, World!
```

Зміст коментарю # Enter зрозумілий і у подальшому не потрібний.

Наведені далі приклади, крім демонстрації простих за змістом обчислень, демонструють й особливість мови **Python**. У мовах високого рівня (C++ і т. і.) програміст відповідає за обов'язкове визначення, перед першим використанням у програмі, типу об'єкту й відповідність змісту виразу (контекст) мови, що контролюється компілятором.

На відміну від цього, при використанні мови надвисокого рівня програміст зосереджується на змісті сценарію. Коректність синтаксису й контексту виразів контролюється інтерпретатором. Тип об'єкту визначається автоматично при трансляції за написом виразу. І якщо вираз правильний, за синтаксисом й контекстом, інтерпретатор обчислює значення виразу.

У всіх прикладах синтаксис правильний. Контекст перших виразів: “додати два числа”, “склеїти списки”, “склеїти рядки”.

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

>>> 'I ' + 'like ' + 'pasta' + '!'
'I like pasta!'
```

Хоча тип об'єктів не визначений явно, інтерпретатор самостійно розібрався у контексті виразів і правильно виконав те, що мав на увазі програміст. У мовах високого рівня подібне неможливе.

У наступному прикладі окрім виразів правильні за написом, але вираз загалом не має сенсу, оскільки зроблена спроба склеїти або додати рядок і ціле число.

```
>>> '42' + 5
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    '42' + 5
TypeError: can only concatenate str (not "int") to str
```

При введенні невірної інструкції, інтерпретатор коментує помилку. У наведених прикладах синтаксис виразу правильний. Проте, інтерпретатор вказує на помилку - відсутність змісту виразу.

Ці ж приклади ілюструють вказану вище особливість мови надвисокого рівня. **Типи даних не визначені програмістом. Інтерпретатор розпізнає їх самостійно. Більш того, в залежності від результатів розпізнавання, по різному коментує помилку.**

Крім операції додання “+”, Python може виконувати наступні операції з числами (таблиця).

Таблиця. Операції з числами у порядку зменшення пріоритету.

Оператор	Символ	Приклад	Результат
Піднесення до степені	**	3.1**2	9.61
Лишок ділення за модулем	%	34%5	4
Ділення із відкиданням лишку	//	34//5	6
Ділення	/	34/5	6.8

Множення	*	$34,1 * 5$	170.5
Віднімання	-	$34.1 - 5.2$	28.9
Додавання	+	$34.1 + 5.2$	39.3

Для зміни пріоритету операції, як і в математиці, використовують дужки

```
>>> (5 + 7)*4
48
>>> 3 + 7*4
31
```

Базові (неділимі) типи даних: цілі числа, дійсні, рядки, логічний (булевий)

Поняття «тип даних» у програмуванні означає з одного боку, тип виразів мови, що використовуються у програмі для опису цих даних. З іншого боку, визначає розміри й структуру області пам'яті, де розміщаються дані, а також методи які можна застосовувати для їх оброблення.

При використанні мови високого рівня, програміст задає змінну, яка є логічним іменем області даних, і атрибут цієї змінної – тип даних, які будуть розміщені у області з таким іменем.

Мова високого рівня надає програмісту можливість отримувати доступ до області даних і безпосередньо за її адресом у пам'яті комп'ютера, використовуючи покажчики. Проте, покажчик все **одно є спеціальною змінною, при визначені якої необхідно вказувати атрибут – тип області даних, на яку він вказуватиме.**

Отже, відповідність за правильне визначення типу даних, що оброблятимуться і створюватимуться програмою, типу області даних у пам'яті комп'ютера і методів оброблення лежить на людині, на програмісті. Як встановлено, із зростанням складності даних та програм продуктивність праці програміста спадає за експоненціальним законом.

Інтерпретатор мови надвисокого рівня **за написом виразу**, що описує дані, самостійно визначає тип цих даних і створює у пам'яті комп'ютера відповідний об'єкт. **Якщо програміст іменує цей вираз у програмі, то значенням цього імені стає адрес об'єкту.** Приклад:

```
>>> 2 # Інтерпретатор за написом визначає, що це дані типу «циле
число» і створює у пам'яті об'єкт з відповідною структурою
2
>>> id(2) # інструкція інтерпретатору на виведення ідентифікатора
140717018126016
```

У такому діалогу програміст до 2 може отримати доступ тільки через ідентифікатор.

```
>>> import ctypes  
>>> print(ctypes.cast(_, ctypes.py_object).value)  
2
```

Але наскільки це зручно, легко бачити. Природно, що доступ до об'єкту у Python, як у всіх мовах високого рівня, забезпечується за допомогою змінних, які є логічним ім'ям об'єкту. А ідентифікатор – фізичне ім'я, оскільки це адреса об'єкту у пам'яті комп'ютера.

```
>>> x=2  
>>> id(x)  
140717018126016
```

Природно, що адреса об'єкту, зрозуміло, не змінилася.

```
>>>> import math >>>  
math.sqrt(x)  
1.4142135623730951
```

Розуміння програмістом цих відмін в архітектурі мов HLL та VHLL має величезне значення і тому варто їх проілюструвати більш докладним прикладом.

Незадовіючи повторимо, у мовах високого рівня область пам'яті, де розташовані дані певного типу, іменується змінною. При розробленні програми програміст **обов'язково** повинен указати компілятору тип області даних у вигляді атрибуту змінної ‘int’, ‘float’ і т. і.

Вираз мови **Python**, синтаксично і змістово правильна послідовність слів мови. За його написом інтерпретатор встановлює, що це дані і їх тип, або що це інструкція і виконує її, тобто всі операції і дії, передбачені прагматикою символів і лексем виразу.

Базові типи мови Python

Базові типи мови **Python** називають ще неділимими, оскільки всі дані є об'єктами і спроба змінити такий об'єкт веде до створення нового об'єкту. Продовжимо попередній приклад:

```
>>> 2  
2  
>>> id(2)  
140712867403456  
>>> x = 2  
>>> id(x)  
140712867403456  
>>> x=x+1  
>>> x  
3  
>>> id(x)  
140712867403488  
>>> id(2)  
140712867403456
```

Легко бачити, при спробі змінити об'єкт базового типу утворюється новий об'єкт і змінна тепер вказує на нього. А вихідний об'єкт, 2, не змінився. **Що на нього вказує, оскільки змінні х вказує на новий об'єкт?** Питання на п'ятірку.

Деякі базові типи даних Python наведені в таблиці нижче. Вирази і семантика даних базових типів мови Python нічим не відрізняються від аналогічних типів у інших мовах високого рівня. О значеннях -2 і 30 говорять, що вони є «цілі числа». Цілочисловому (int) типу даних відповідають значення у вигляді цілих чисел. Числа десятковою точкою, такі, наприклад, як 3.14, називаються числами з плаваючою точкою (тип даних float) або дійсними числами. Єдине треба зауважити, що значення змінних булевого типу, константи True і False, пишуться з великої літери **обов'язково**.

Таблиця. Базові типи даних Python

Базові типи даних	Приклади значень змінних цього типу
Цілі числа (int)	-2, -1, 0, 1, 2, 3, 4, 5
Дійсні числа (float)	-1.25, -1.0, -0.5, 0.0, 0.5, 1.0, 1.25
Рядки (str)	'a', 'Hello!', '', 'I like pasta', '3.14'
Логічний (bool)	True, False

Програмуючи мовою Python, можна використовувати і текстові значення, так звані рядки (базовий тип даних str). Синтаксис рядка, за яким інтерпретатор розпізнає вираз як рядок, це будь яка послідовність символів алфавіту (у тому числі й пробіл), між одинарними (або подвійними) лапками.

Наприклад:

'Hello' або 'Good bye, cruel world! " .

Рядок може взагалі не містити жодного символу (''). Такі рядки називаються порожніми.

Найбільш поширена помилка при введені рядків, відсутність апострофа наприкінці рядку.

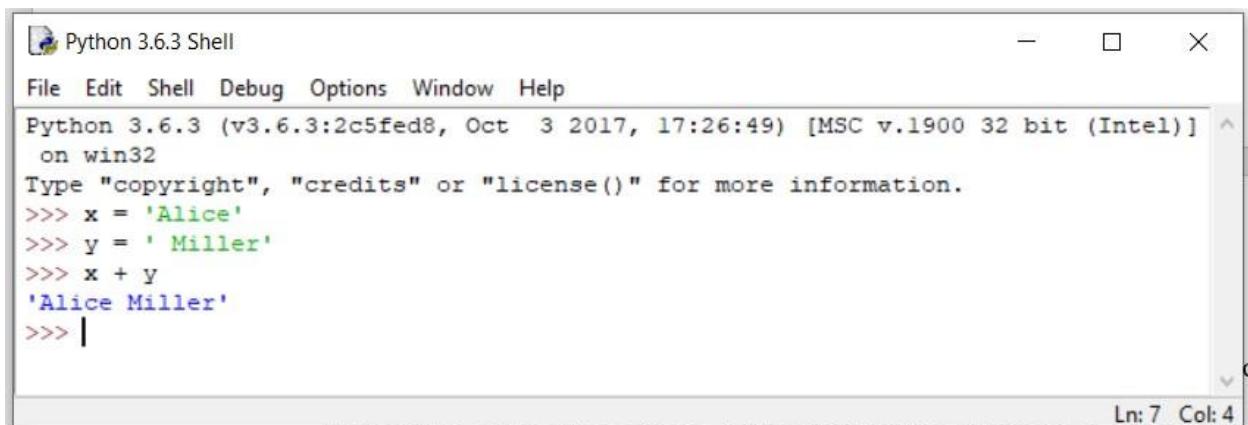
Наприклад:

```
>>> 'Hello, world!
SyntaxError: EOL while scanning string literal
>>>
```

Конкатенація й реплікація рядків

Як говорилося вище, інтерпретатор мови надвисокого рівня розпізнає вирази не тільки за синтаксисом, але й і за контекстом. І якщо контекст має зміст, інтерпретатор обчислює значення виразу.

Вираз вигляду ‘послідовність1 символів’ + ‘послідовність2 символів’ має зміст «склеїти рядки» і інтерпретатор обчислює значення виразу. Приклад:



The screenshot shows a Python 3.6.3 Shell window. The title bar says "Python 3.6.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the Python interpreter's prompt and some code. The code consists of four lines: "x = 'Alice'", "y = ' Miller'", "x + y", and an empty line. The output is "'Alice Miller'". In the bottom right corner of the shell window, there is a status bar with "Ln: 7 Col: 4".

```
Python 3.6.3 (v3.6.3:2c5fed8, Oct  3 2017, 17:26:49) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> x = 'Alice'
>>> y = ' Miller'
>>> x + y
'Alice Miller'
>>> |
```

Така операція часто використовується при обробленні рядків і називається «конкатенацією».

Перший вираз у наведеному нижче прикладі змісту не має,

```
>>> (x + y) + 5
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    (x + y) + 5
TypeError: must be str, not int
>>> (x + y) * 5
'Alice MillerAlice MillerAlice MillerAlice MillerAlice Miller'
>>> |
```

У цьому ж прикладі показаний інший вираз, цілком зрозумілий і він інтерпретується правильно. Така операція називається «реплікацією»

Імена об'єктів програми

Області пам'яті комп'ютера, де розміщені дані, адресуються, щоб програма мала доступ до них.

У внутрішньому поданні мови програмування адрес має вигляд числа, наприклад, шістнадцятирічного (C++). Оскільки таке подання адресу незручне, програміст іменує дані. Імена (ідентифікатори), незалежно від типу даних (змінні, списки, функції і т. і.), мають вигляд виразу, синтаксис якого підпорядкований певним правилам, практично однаковим для мов високого рівня:

1. Ім'я повинно бути одним словом.
2. В імені можуть використовуватися тільки букви, цифри і символ підкреслювання (_).
3. Ім'я не може починатися із цифри.

Таблиця. Припустимі й неприпустимі імена даних

Припустимі імена	Неприпустимі імена
Balapce_current	Balance-current
Ba1anceCurrent	Balance current
Aim aim	4account
aIm	

У мова надвисокого рівня імена чутливі до регістру. Імена Aim, aim і aIm (Табл.) різні у мові Python.

Загально відомо, що при іменуванні об'єктів програми ідентифікатори слід вибирати так, щоб вони несли і змістовне навантаження. Якщо зміст складається із декілька слів, використовується нижнє підкреслювання (таблиця).

Документ **PEP 8** (<https://peps.python.org/pep-0008/>) рекомендує і так званий верблюжий стиль

CaMeL

Нижнє підкреслювання використовувати не рекомендується.

Про стиль написання коду програми написано багато. Це рекомендації, накопичений досвід програмістів, спрямовані на підвищення продуктивності роботи із кодом на протязі всього життєвого циклу програми.

У зв'язку з цим, розробники Python приділяють велике значення стилю написання коду. Цей процес навіть стандартизований документом **PEP 8**. Професійний стиль написання коду є синтаксичними правилами мови **Python**. Отже, користувачі Python **вимушенні** професійно відноситися до написання коду. Порушення стилю інтерпретатор сприймає як синтаксичну помилку у тексті програми. Красиве рішення!

Приклад.

Одним з найважливіших є правило форматування коду на підставі парадигми структурного програмування.

Незалежно від суті алгоритму, стилю та мови програмування, програму завжди можна представити у вигляді ієархії блоків управління, кожний з яких є композицією лише трьох блоків управління – слідування, розгалуження та циклу. Це твердження є точним математичним фактом (теорема Бьюма-Якопіні). Саме тому будь-яка мова програмування містить спеціальні символи, так звані операторні дужки, які обмежують ці блоки. Вони виглядають по різному: “begin end”, “{ }”, “if ... fi”, “do ... od” і т. і. Проте, є обов'язковими.

Професійний стиль написання коду полягає у тому, що блоки у тексті програміст виділяє блоки різною кількістю відступів. Проте, це не є обов'язковим у всіх мовах, крім **Python**. Програміст вимушений писати «красиво». «Некрасивий текст» буде сприйнятий інтерпретатором, як синтаксично не правильній.

Коментарі

Коментарі у HLL й VHLL нічим не відрізняються. - це частина тексту програми, яку ігнорує компілятор і інтерпретатор. Python ігнорує коментарі – будький текст від символу # до кінця рядка. Коментарі у програмі, як правило:

1. Пояснюють логіку і структуру програми або алгоритму та іншої корисної інформації. Якість коментування програми вважається загальновизнаною ознакою кваліфікації програміста.

2. Засіб тимчасового ігнорування інтерпретатором рядка коду програми. Ефективність такого простого прийому налагодження програми загально відома. Порожні рядки є також ігноруються інтерпретатором. Їх використовують для форматування тексту програм, що також вважається добрым стилем програмування.

Наскільки коментарі важливі і як багато їх повинно бути у тексті програми. Наведемо думку одного із провідних програмістів 20-го сторіччя Д. Ван-Тассела: «Коментарів у тексті програми завжди повинно бути більше, ніж вам це здається».

Деякі функції, необхідні на перших кроках

Ядро містить цілий ряд функцій, необхідних інтерактивної роботи користувача. Наведемо деякі?

Функція print()

Функція `print('рядок')` виводить на екран **рядок**, який є аргументом функції. Лапки при цьому не виводяться. Аргументом функції може бути і порожній рядок `print()`.

Функція input()

Функція `input()` очікує введення користувачем виразу з подальшим натисненням клавіши <Enter>:

```
>>> myName= input()
Alex
>>> myName
'Alex'
>>> |
```

Функція `input()` перетворює текст, введений користувачем, **і завжди повертає рядок**, який стає значенням змінної, яка іменує введення (`myName` у прикладі). Значенням змінної стає введений рядок. У прикладі значенням `myName` стає об'єкт 'Alex'.

Функція input() завжди повертає рядок.

```
>>> n = input()
45
>>> 55 + n
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    55 + n
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> type(n)
<class 'str'>
>>> |
```

Якщо користувачу треба ввести дані іншого типу, використовуються функції перетворення типу (див. далі).

Функція len()

За необхідності можна передати функції `len()` строковий вираз і вона поверне ціле число, рівне кількості символів у даному рядку, що і виконує наступний код програми на рисунку нижче.

Перетворення базових типів. Функції str(), int(), float() .

Аргументами функцій `input()`, `print()` та `length()` є дані строкового типу. Проте, на практиці часто виникає необхідність або вводити у режимі діалогу, або

друкувати дані, які мають належать до іншого типу даних. Наступні приклади моделюють подібні ситуації:

```
>>> v = 'I am ' + ' a man '
>>> n = len(v)
>>> print('length of exprssion v = ' + v + 'is' + n)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print('length of exprssion v = ' + v + 'is' + n)
TypeError: can only concatenate str (not "int") to str
>>> print('length of exprssion v = ' + v + 'is ' + str(n))
length of exprssion v = I am  a man is 12
>>>
```

Помилка пов'язана не з самими функціями `input()` та `print()`, а з типом виразів, що намагаються їм передати. Після перетворення даних все нормальню.

Функції `str()`, `int()` та `float()` дають змогу усунути цю невідповідність, перетворивши тип аргументу. Функції `str()`, `int()` та `float()` відповідно повертають рядок, ціле число і дійсне число, якщо їх можна поставити у відповідність аргументу:

```
>>> str(54)
'54'
>>> str(-1.32)
'-1.32'
>>> int(-1.32)
-1
>>> int('ab')
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    int('ab')
ValueError: invalid literal for int() with base 10: 'ab'
>>> float(54)
54.0
>>> float('ab')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    float('ab')
ValueError: could not convert string to float: 'ab'
>>> str(5, 'utf-16')
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    str(5, 'utf-16')
TypeError: decoding to str: need a bytes-like object, int found
>>> str(5)
'5'
```

У цих прикладах функція `str()` викликається для отримання рядкової, а `int()` та `float()` цілочисленної та дійсної форм значень, поданих іншими типами даних, якщо це має зміст.

Досить часто функцію `str()` зручно використовувати у тих випадках, коли є ціле або дійсне число, яке треба вклейти у рядок. Функція `int()` буде корисною,

якщо є число у рядковій формі (наприклад, при введені функцією `input()`, див. вище), яке необхідно використовувати у математичних обчисленнях.

```
>>> b = input()
3.14
>>> b/2
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    b/2
TypeError: unsupported operand type(s) for /: 'str' and 'int'
>>> # Проте
>>> b = float(input())
3.14
>>> b/2
1.57
>>> |
```

Треба ще додати, що функцію `int()` зручно використовувати для округлення дійсних чисел (див. приклад).

Контрольні запитання

1. Які з наведених нижче синтаксичних елементів являють собою операції, а як об'єкти * 'hello'
-88.8
-
/
+
5
2. Чим відрізняються вирази мови Python `myName` і '`myName`'?
3. Назвіть та дайте характеристику базовим типам даних мови Python 4. Що таке вираз мови Python? Наведіть приклади.
5. Чим відрізняються поняття змінної мови високого рівня та надвисокого рівня. Продемонструйте прикладами.
6. Чи буде змінюватися тип змінної `var` у процесі виконання коду

```
>>> var = 20
>>> type(var)
<class 'int'>
>>> var = var / 3.1
>>> type(var) # ????
```

7. Чи мають зміст вирази?

‘Bekki’ + ‘Tom’
‘Bekki’ * 3 + ‘Tom’ * 3 ('Bekki'
+ ‘Tom’) * 3

8. Наведіть приклади припустимих та не припустимих виразів, що іменують об'єкти у мові Python.
9. Яке значення повертають функції **int()**, **float()**, **str()** в залежності від значення аргументу? Скласти таблицю.
10. Яким буде результат виконання інструкції

```
|>>> 'I' + 'ate' + 12 + 'slapjack' # ???|
```

11. Анонімна змінна, це автоматично утворене посилання на область пам'яті, де розташований останній результат обчислень, якщо він не має ідентифікатор. Які дії відбувалися за цим лістингом

```
>>> import math  
>>> math.pi  
3.141592653589793  
>>>  
3.141592653589793  
>>> id(_)  
1777763483184  
>>>  
1777763483184  
>>>
```

12. Чому результати майже однакових кодів різні?

```
>>> b = a =[1,2]  
>>> a[0]=2  
>>> b  
[2, 2]  
>>> |
```

Й

```
>>> b = a = [1,2]  
>>> a = [2,2]  
>>> b  
[1, 2]  
>>> |
```

Завдання 1

1. Використовуючи командний рядок ОС, упевнитися, що бібліотеки **math**, **matplotlib**, **scipy** завантажені. Якщо їх немає у переліку завантажених – встановити.
2. Бібліотека математичних функцій **math**

Ядро **Python** містить деякі математичні функції та операції. Спеціальна математична бібліотека **math** містить широкий спектр математичних функцій. Функції, аналогічні тим, що є у ядрі, також присутні. Проте виконуються вони швидше, оптимізовано і з більшою точністю.

Приклад: Обчислення кореня квадратного з числа 2.

```
>>> pow(2,1/2)
1.4142135623730951
>>> 2** (1/2)
1.4142135623730951
>>> math.sqrt(2)
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    math.sqrt(2)
NameError: name 'math' is not defined
>>> import math
>>> math.sqrt(2)
1.4142135623730951
>>> math.pow(2,1/2)
1.4142135623730951
>>>
```

У прикладі, спочатку застосовуються функції ядра. Потім застосовується функція бібліотеки. Проте, інтерпретатор видав помилку, оскільки бібліотека не імпортована.

Завдання 1. Обчислити: А)

Обчислити:

$$2 \underline{\quad\quad\quad} 3^{-3} 2^5 5^3 2^{10} 0.5^{-2}^4 ; \quad \sqrt[5]{2180}$$

Б) Обчислити, надавши змінним два різних значення:

$$\begin{aligned} z_1 &= 2 \sin^2(3\pi - 2\alpha) \cos^2(5\pi + 2\alpha) \\ z^2 &= \frac{1}{4} - \frac{1}{4} \cos\left(\frac{5}{2}\pi - 8\alpha\right) \\ z &= \frac{a+b}{\sqrt{3}(a-b)} \end{aligned}$$

3. Дослідити у режимі калькулятора всі можливості функцій `input()`, `print()`, використовуючи інформацію з веб-ресурсів.
4. Бібліотека **matplotlib** містить функції графіки. Бібліотека **scipy** містить різноманітні функції для наукових та інженерних розрахунків. Бібліотека **numpy** містить функції для формування структур даних (вектори, матриці, масиви тощо).

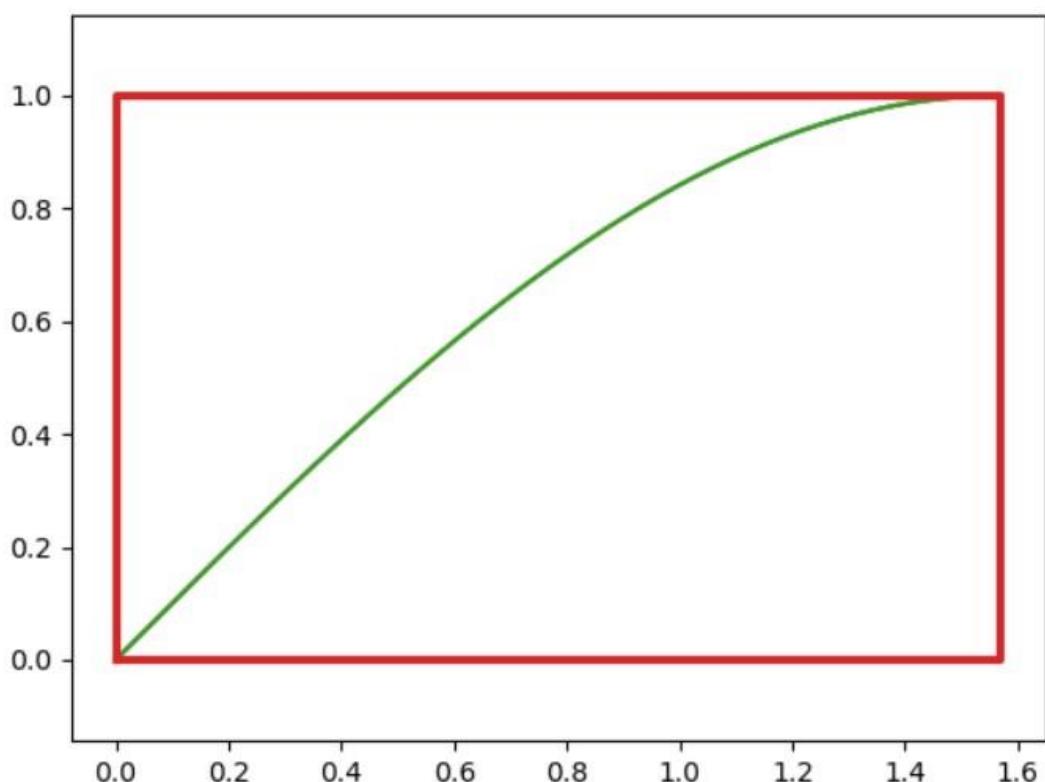
Приклад: На які за величиною площі ділить синусоїда прямокутник на рисунку

```

>>> import numpy as np, matplotlib.pyplot as plt, math, scipy
>>> X = np.linspace(0, math.pi/2, 50)
>>> Y = np.sin(X)
>>> plt.plot(X, Y)
[<matplotlib.lines.Line2D object at 0x000001ABA3B7F820>]
>>> plt.plot([0,0,math.pi/2,math.pi/2,0],[0,1,1,0,0], linewidth=2)
[<matplotlib.lines.Line2D object at 0x000001ABA3B7FC40>]
>>> plt.axis('equal')
(-0.07853981633974483, 1.6493361431346414, -0.05, 1.05)
>>> plt.show()

```

Figure 1



x=0.674 y=0.003

Обчислення площ за допомогою визначеного інтегралу, реалізованого функцією у бібліотеці **scipy** Лістинг:

```

>>> from scipy.integrate import quad
>>> Str = 1 * math.pi/2
>>> I = quad(math.sin, 0, math.pi/2)
>>> I
(0.9999999999999999, 1.1102230246251564e-14)
>>> S1=I[0]
>>> S2 = Str - S1
>>> print('Square above curve is ', S1, ' under curve is ', S2)
Square above curve is  0.9999999999999999  under curve is  0.5707963267948967
>>>

```

Послідовність дій:

1. Завантаження функції чисельного обчислення визначеного інтегралу.
2. Обчислення площі прямокутника
3. Обчислення визначеного інтегралу. Функція повертає список із значення інтегралу й точності обчислення.
4. Перший елемент списку – площа криволінійної трапеції під кривою.
5. Площа над кривою обчислена як різниця площ прямокутника й площі під кривою.

Завдання 2:

1. Побудувати графіки функцій $y = x^2$, $y = \sqrt{x}$. Знайти площу фігури, обмежену цими лініями.
2. Дано прямокутник з вершинами A(1,0), B(1, 1), C(e , 1), D(e , 0) й функція $y = \ln x$.

Знайти площі фігур, на які ділить графік функції прямокутник.

Інформаційний ресурс:

1. <https://scipy.org/about.html>
2. <https://www.math.ubc.ca/~pwalls/math-python/scipy/scipy/>

Лабораторні роботи 5, 6, 7 **Тема:** «Потоки управління»

Мета:

1. Познайомитися з роботою інтерпретатора Python у режимі програмування.
2. Поновити та отримати знання із структурного програмування.
3. Відповісти на контрольні питання та виконати завдання.

Ваша перша програма

Інтерактивна оболонка відмінно годиться до виконання інструкцій Python по одній за один раз, але при написанні повноцінних програм на Python використовуються текстові редактори, такі як Notepad++, TexxtMatc та інші. Оболонка IDLE має власний редактор.

Якщо вибрати пункти меню File -> New file, то відкриється вікно та курсор, що очікує введення. Проте, це вікно відрізняється від вікна інтерактивної оболонки, яке виконує введену інструкцію Python сразу ж після натискання клавіши <Enter>. Текстовий редактор дає змогу ввести множину інструкцій, зберегти файл і виконати програму.

 Python 3.7.4 Shell

- □ ×

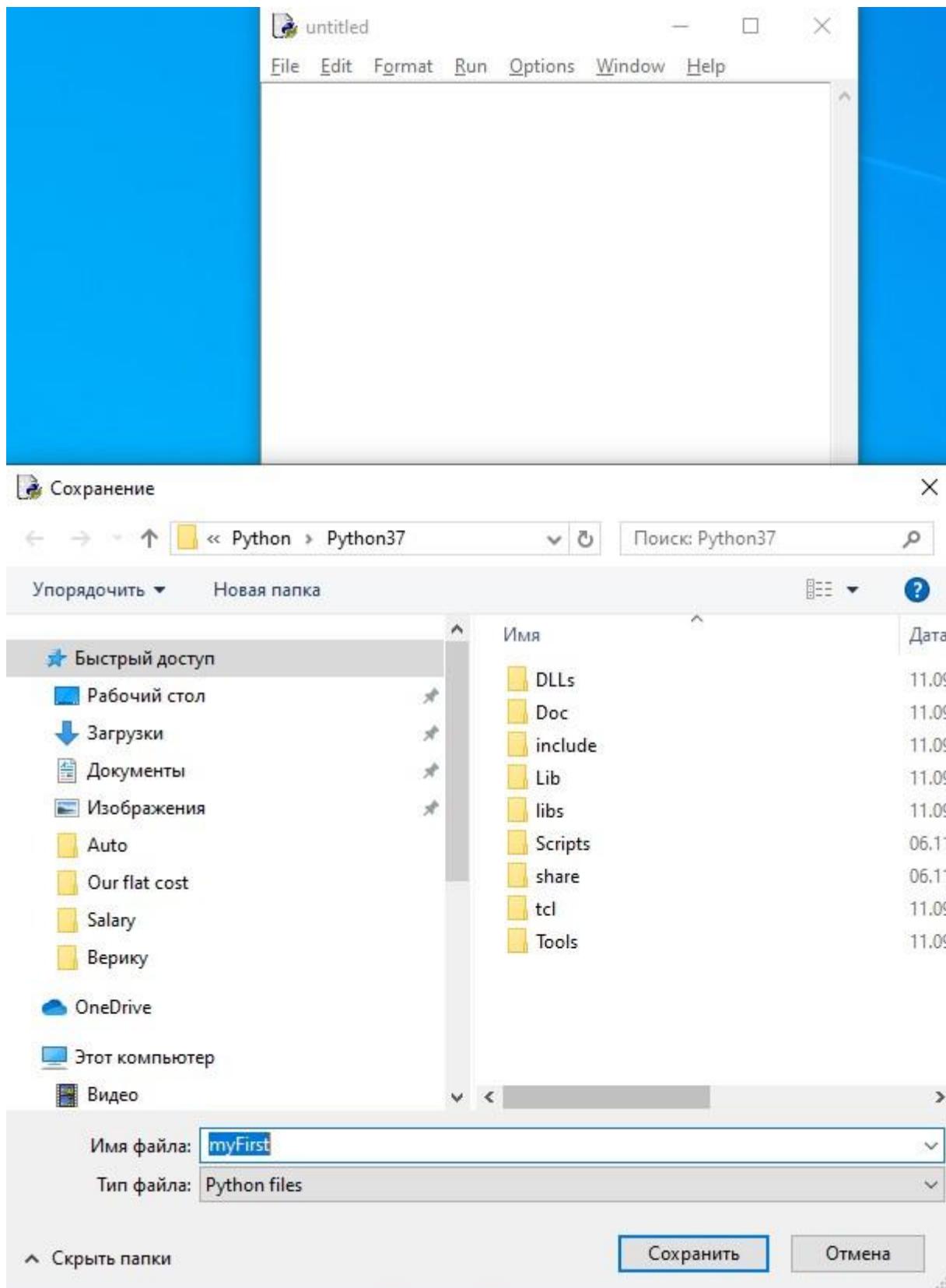
File Edit Shell Debug Options Window Help

- New File Ctrl+N
- Open... Ctrl+O
- Open Module... Alt+M
- Recent Files
- Module Browser Alt+C
- Path Browser
- Save Ctrl+S
- Save As... Ctrl+Shift+S
- Save Copy As... Alt+Shift+S
- Print Window Ctrl+P
- Close Alt+F4
- Exit Ctrl+Q

:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit]

"credits" or "license()" for more information.

Ln: 3 Col: 4



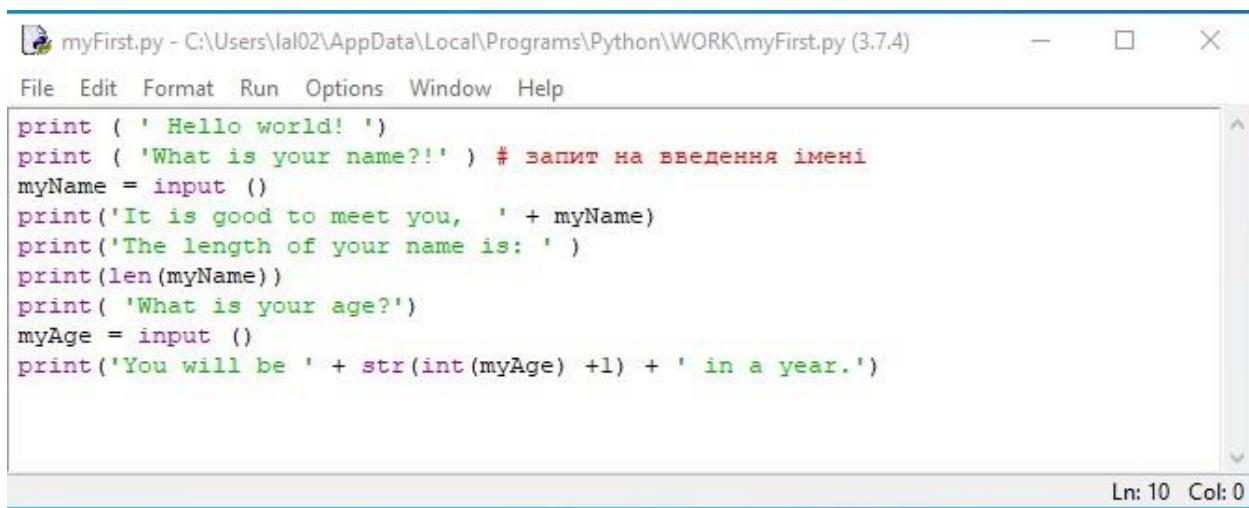
Нижче вказано, чим розрізняються ці два вікна:

- ознакою вікна інтерактивної оболонки є запрошення до введення коду >>>;
- у вікні файлового редактора запрошення до введення >>> відсутнє.

Відкриваємо вікно текстового редактора, вводимо текст програми.

По введенню коду, зберегаємо його, щоб не набирати його заново при наступному сеансі із IDLE. Вибираємо у меню, розташованому у верхній частині вікна текстового редактору, пункти File -> Save As, введіть ім'я файлу, наприклад myFirst у полі File Name й клікніть Save . Файл отримає за замовчанням розширення і буде збережений з повним іменем myFirst.py у робочу папку.

У процесі введення тексту програми періодично зберігайте файл, що дозволить уникнути втрати вже введеного коду, якщо робота оболонки IDLE буде завершена несанкціоновано.

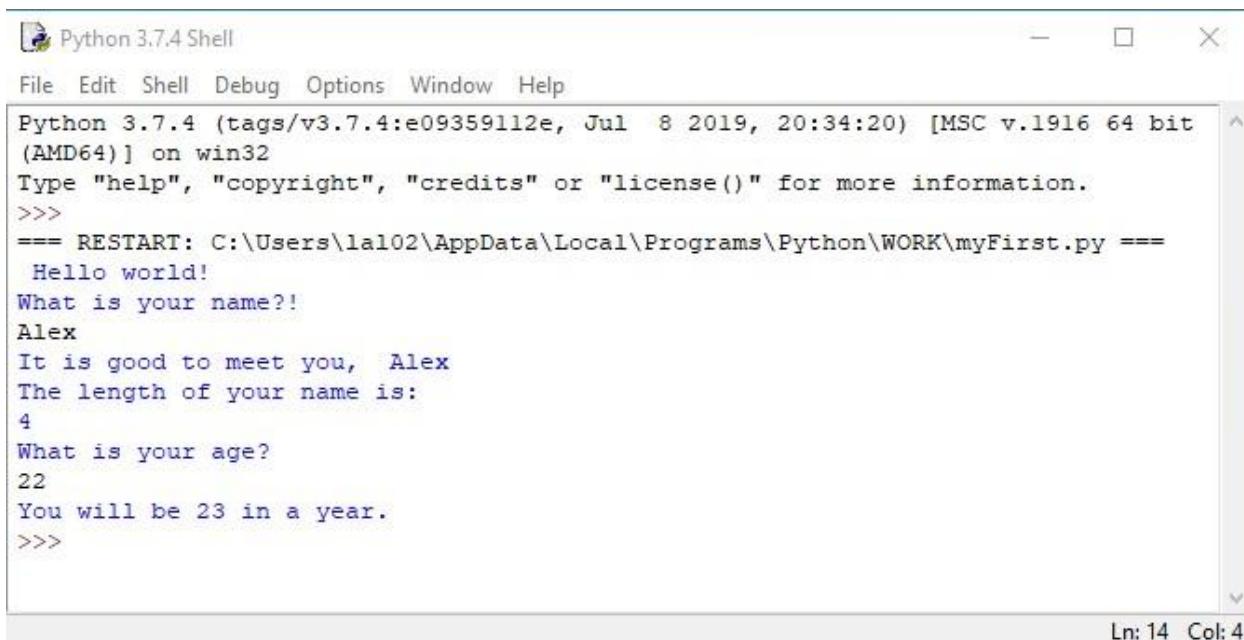


```
myFirst.py - C:\Users\lal02\AppData\Local\Programs\Python\WORK\myFirst.py (3.7.4)
File Edit Format Run Options Window Help
print ('Hello world!')
print ('What is your name?') # запит на введення імені
myName = input ()
print('It is good to meet you, ' + myName)
print('The length of your name is: ')
print(len(myName))
print('What is your age?')
myAge = input ()
print('You will be ' + str(int(myAge) +1) + ' in a year.')

Ln: 10 Col: 0
```

Після того як файл буде збережено, запустіть програму. Виберіть пункти меню Run -> Run Module або просто натисніть клавішу <F5>. Ваша програма повинна запуститься у вікні інтерактивної оболонки, яке відкривалося, коли вперше запускали IDLE. Клавішу <F5> слід натискати не у вікні інтерактивної оболонки, а у вікні файлового редактора.

Введіть своє ім'я у відповідь на запрошення програми. Вивід програми у вікні інтерактивної оболонки повинен виглядати приблизно так:



Python 3.7.4 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.4 (tags/v3.7.4:9025b0b, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
== RESTART: C:\Users\lal02\AppData\Local\Programs\Python\WORK\myFirst.py ===
Hello world!
What is your name?!
Alex
It is good to meet you, Alex
The length of your name is:
4
What is your age?
22
You will be 23 in a year.
>>>
```

Ln: 14 Col: 4

Вичерпавши всі рядки коду програма завершується.

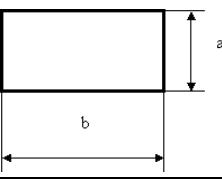
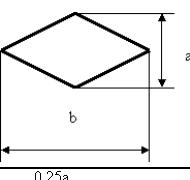
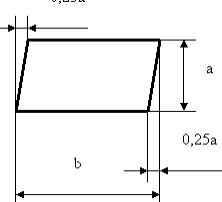
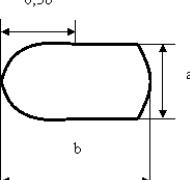
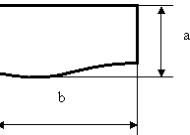
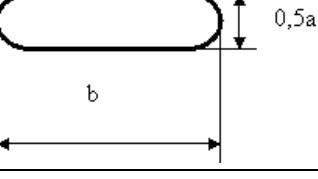
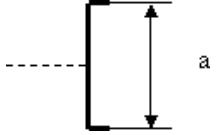
Теоретичний мінімум

Структурне програмування – це парадигма програмування в основі котрої лежить уявлення про програму, як ієархічну структуру, утворену функціональними блоків, а процес викання полягає у виконанні кожним блоком своєї функції і передачі отриманих результатів та управління наступному блоку, у відповідності із логікою програми.

Послідовність передачі управління часто називають потоком управління програми. Розроблення потоку управління є одним із найважливіших етапів проектування ПЗ. При цьому широко використовується різні графічні подання. Ми будемо користуватися блок-схемами.

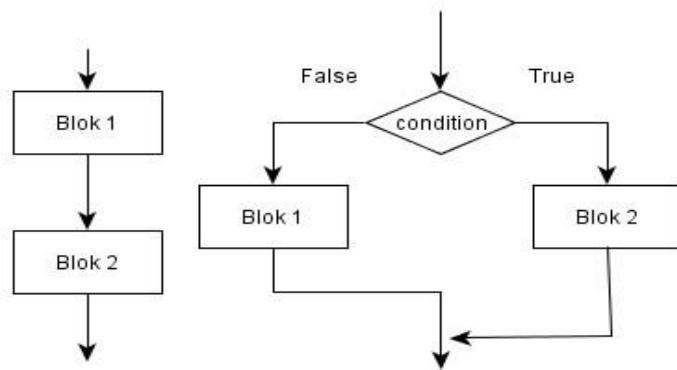
Стандартом зображення блок-схем є **ISO 5807:2016. Зображення елементів блок-схем згідно цього стандарту та основні правила наведені у Додатку**. Тут наведені деякі Докладно Ромбовидні блоки на цих схемах, це блок у якому виконується алгоритм порівняння характеристик даних, оброблених попередніми блоками, з певними цільовими характеристиками цих даних. Блоки, у яких виконуються дії – прямокутниками.

Форма та зміст блоків алгоритму		
Найменування	Позначення	Функції
1	2	3

Процес		Виконання операцій присвоювання, наприклад, $A = 0$, додавання з присвоюванням, на- приклад, $C = A+B$, віднімання, множення і т. і.
Рішення		Вибір напрямку виконання алгоритму (програми) в залежності від деяких змінних умов
Введення - виведення		Введення – виведення даних незалежно від типу пристроя введення або виведення
Дисплей		Введення даних з дисплея (клавіатури), виведення даних на дисплей
Документ		Виведення даних на папір (принтер)
Пуск - зупин		Початок – кінець алгоритму (програми)
Поєднання		Перехід на блок номер 5 (номер блоку наведений для прикладу)
Коментар		-

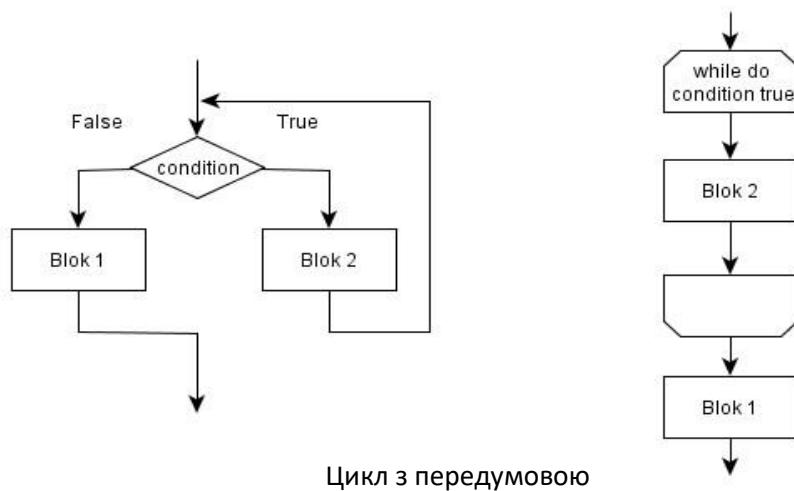
Потік управління програми завжди можна подати як ієрархію та композицію трьох основних структур управління (відома теорема Бъома-Якопіні):

1. Послідовне управління – по завершенню роботи блока, управління безумовно передається наступному блоку.
2. Передача управління за умовою, або гілкування процесу управління. Робота ромбовидного блоку (рішення) полягає у порівнянні значень характеристик результатів роботи попередніх блоків цільовим значенням. В залежності від результатів порівняння, управління передається одному з альтернативних блоків.
3. Ітерації або цикли у процесі управління. З досвіду випливає, що доцільно розглядати такі різновиди циклів, як цикл з передумовою, цикл з пост умовою і цикл з лічильником. Доцільність підтверджена хоча тим, що усі мови високого і вище рівня мають окремі інструкції для кожного різновиду.
 - Цикл ‘while do’ з передумовою (рішення приймається до початку процесу обчислень). Результати роботи попереднього блоку передаються блоку перевірки цільової умови. Якщо «так», управління передається наступному блоку. Як «ні», виконується додатковий блок і, потім знову перевіряється цільова умова.
 - Цикл ‘do while’ з пост умовою (рішення приймається після процесу обчислень). Виконується блок 1 і дані передаються для перевірки цільової умови. Якщо «так», управління передається наступному блоку. Як «ні», управління повертається першому блоку.
 - Цикл ‘for each’ з лічильником. По суті це цикл з передумовою. Різниця полягає у тому, що кількість ітерацій визначається не відповідністю оброблених даних цільовим характеристикам, а кількістю значень рангованої змінної (див далі). Частинним випадком є цикл ‘for’, коли рангованою змінною є індекси. Це настільки поширені випадки, що усі мови високого рівня для них мають окремі інструкції або оператори.

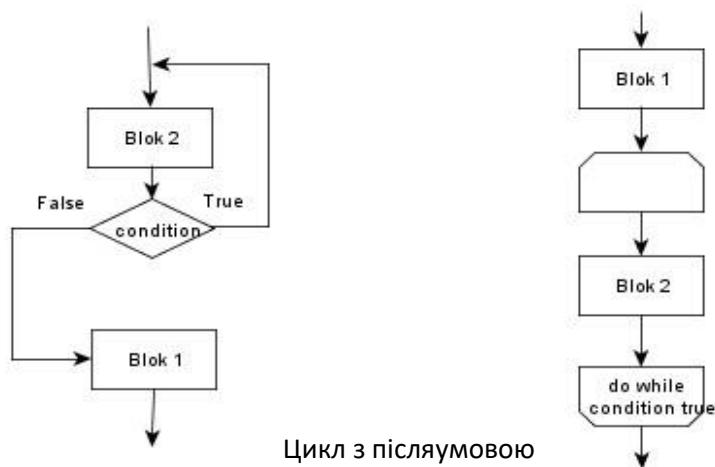


Послідовне управління Управління за умовою

Блок-схеми циклічних структур досить громіздкі, стандарт передбачає скорочені позначення.



Цикл з передумовою



Цикл з післяумовою

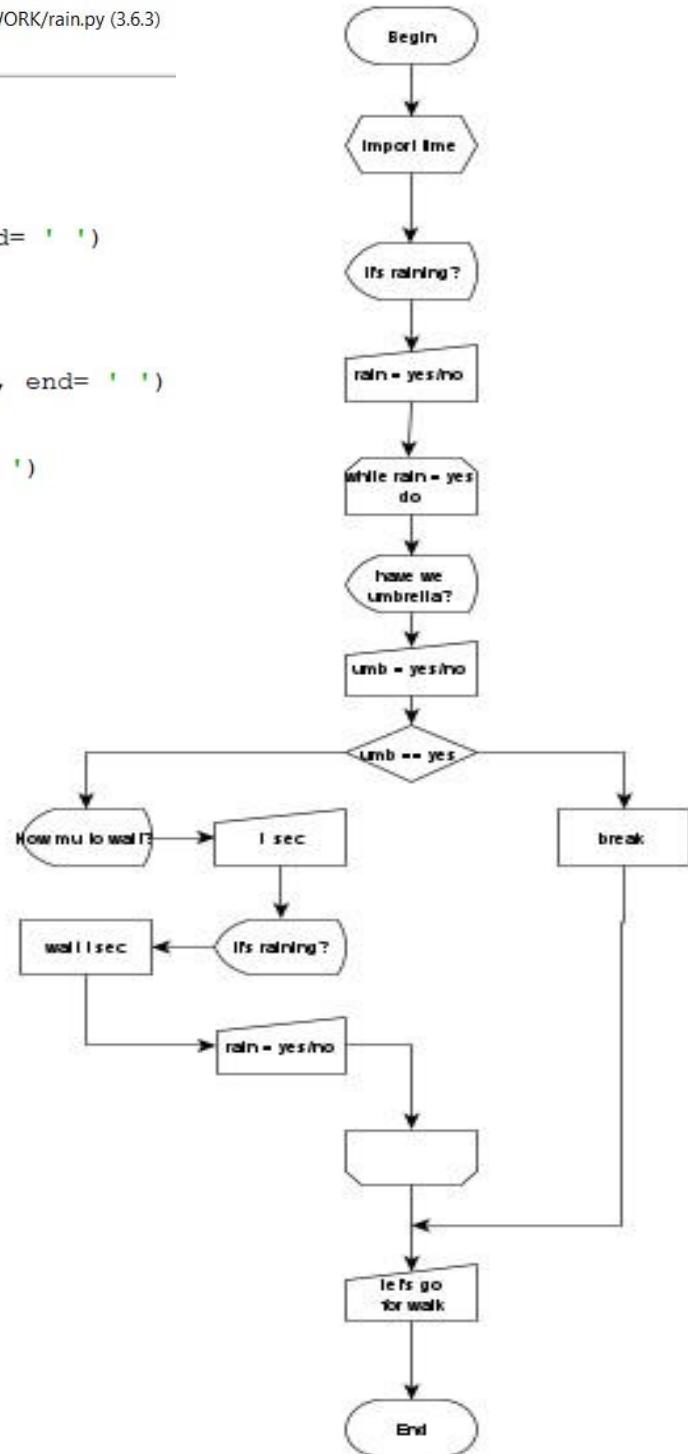
Нижче наведений приклад потоку управління, який є композицією основних структур.

```
rain.py - C:/Users/Alex/AppData/Local/Programs/Python/WORK/rain.py (3.6.3)
File Edit Format Run Options Window Help
import time
print('it''s raining', end= ' ')
rain = input()

while rain == 'yes':
    print('Have we umbrella?', end= ' ')
    umb = input()
    if umb == 'yes':
        break
    else:
        print('How much to wait?', end= ' ')
        t = int(input())
        time.sleep(t)
    print('it''s raining', end= ' ')
    rain = input()

print('lets go for walking')

===== RESTART: C:/Users/Alex/WORK/rain.py =====
it's raining yes
Have we umbrella? no
How much to wait? 5
its raining yes
Have we umbrella? no
How much to wait? 2
its raining no
lets go for walking
>>>
```



Діалог й прийняття рішення
щодо можливості прогулянки

Мова Python містить всі необхідні інструкції передачі управління. Проте, перед тим, як вивчати інструкції управління треба познайомитися зі способами подання мовою Python умов передачі управління. У наведеному прикладі умови

передачі управління – це висловлювання, які можуть приймати значення True або False

Алгебра висловлювань мовою Python

Умови передачі управління мають вигляд висловлювань алгебри логіки, значеннями яких є булеві константи True або False. Вирази є записом алгоритму операцій порівняння та булевих операцій над даними.

Булеві значення та операції

У той час як цілий, дійсний та рядковий типи даних можуть мати практично необмежену кількість можливих значень, логічний, або булевий, тип даних може приймати тільки два значення – константи **True** і **False**. При використанні у коді **Python** булеві значення **True** й **False** завжди пишуться з великої літери.

Приклади виразів мовою **Python** (деякі навмисно записані з типовими помилками):

```
>>> True; False; type(True); type(False)    #1.  
True  
False  
<class 'bool'>  
<class 'bool'>  
>>> var = True; type(var)    # 2.  
<class 'bool'>  
>>> var  
True  
>>> true = 22; type(true)    # 3.  
<class 'int'>  
>>> true  
22  
>>> True  
True  
>>>
```

У цих прикладах:

1. **True** й **False** – константи, які вичерпують множину значень змінних типу ‘**bool**’.
2. Створення й ініціалізація змінної логічного типу, яка може приймати значення тільки **True** або **False**.
3. Можна, але не варто утворювати змінні з іменами, схожими на логічні константи.

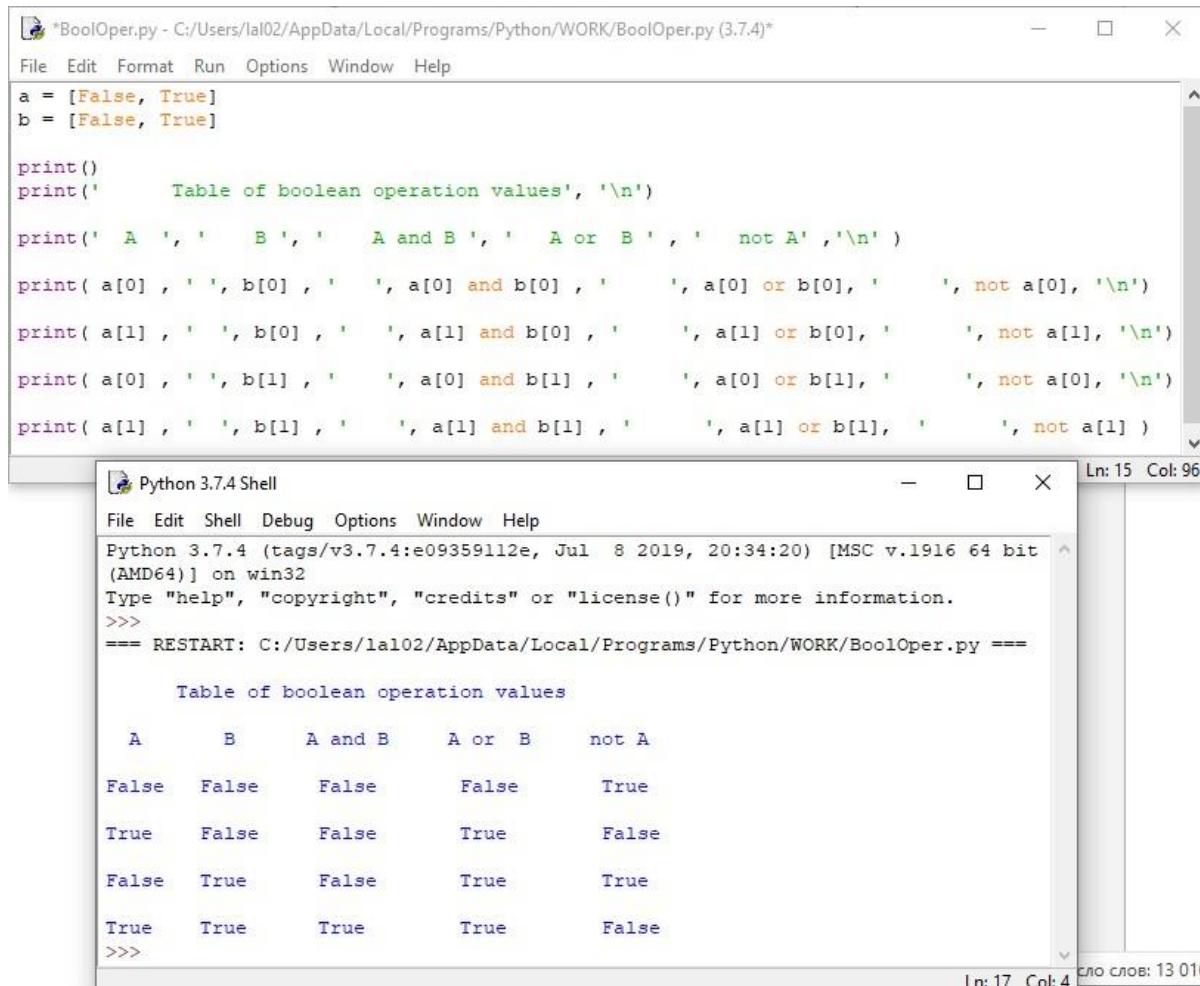
У **Python**, як і в інших мовах програмування високого рівня, для конструювання відповідних виразів реалізована булева алгебра. Як відомо, для запису будь-якого висловлювання у межах цієї алгебри достатньо трьох операцій **and**, **or**, **not**, таблиці істинності яких наведені нижче.

Булеві вирази – один із базових типів даних **Python**.

Символ	Операція булевої алгебри
and	логічне «і»

or	логічне «або»
not	логічне заперечення

Нижче наведений код та результати побудови таблиць істинності операцій булевої алгебри.



The screenshot shows two windows. The top window is titled "BoolOper.py - C:/Users/lal02/AppData/Local/Programs/Python/WORK/BoolOper.py (3.7.4)*". It contains Python code to print a truth table for boolean operations. The bottom window is titled "Python 3.7.4 Shell". It shows the execution of the script, which prints the following truth table:

A	B	A and B	A or B	not A
False	False	False	False	True
True	False	False	True	False
False	True	False	True	True
True	True	True	True	False

Як відомо, булевих операцій (таблиця) достатньо для подання будь-якої іншої операції і запису будь-якого виразу алгебри логіки. На приклад, операція виключної диз'юнкції \oplus (інше позначення XOR) може бути подана формулою

$$a \oplus b = a \text{ and not } b \text{ or not } a \text{ and } b$$

Нижче наведений код та результат побудови таблиці істинності цієї операції:

```

bool.py - C:/Users/Alex/AppData/Local/Programs/Python/WORK/bool.py (3.6.3)
File Edit Format Run Options Window Help
a = [False, True]
b = [False, True]
print('      Table values of boolean operation xor (' , '\u2295', ')', '\n')
print(' *15, 'A', '*5, 'B', '*5, 'A ', '\u2295', ' B', '\n')
print(' *13, a[0], ' , b[0], '*4, a[0] and not b[0] or not a[0] and b[0], '\n')
print(' *13, a[0], ' , b[1], '*5, a[0] and not b[1] or not a[0] and b[1], '\n')
print(' *13, a[1], ' , b[0], '*4, a[1] and not b[0] or not a[1] and b[0], '\n')
print(' *13, a[1], ' , b[1], '*5, a[1] and not b[1] or not a[1] and b[1], '\n')

```

```

===== RESTART: C:/Users/Alex/AppData/Local/Programs/Python/WORK/bool.py
Table values of boolean operation xor ( * )

```

A	B	A * B
False	False	False
False	True	True
True	False	True
True	True	False

Даний приклад не формальний, оскільки операція виключної диз'юнкції входить у сигнатуру алгебри Жигалкіна, яка є визначеною при моделюванні релейних ланцюгів.

Операції порівняння

Операндами висловлювань є вирази булевої алгебри, які містять операції порівняння. Символи мови **Python** для операцій порівняння наведені у таблиці

Таблиця

Символ	Операція порівняння
<code>==</code>	логічне «рівно»
<code>!=</code>	логічне «не рівно»
<code><</code>	менше ніж
<code>></code>	більше ніж
<code><=</code>	менше або рівно
<code>>=</code>	більше або рівно

Реалізація усіх операцій повністю відповідає властивостям і законам алгебри висловлювань. Операції порівняння бінарні і результатом застосування, в залежності від наданих значений, є константи `True` або `False`. Нижче наведені приклади застосуванні цих операцій.

Порівняння даних здійснюється на основі відношення порядку, яке у Python визначено не тільки на базових типах даних, але й на структурах даних (див. далі).

Приклади:

1. Відношення порядку (таблиця)

```
>>> # Відношення порядку на числах
>>> 2 == 3
False
>>> a = 2 < 3
>>> type(a)
<class 'bool'>
>>> a
True
>>> 2 % 3 != 1
True
>>> #
>>> # Відношення лексикографічної упорядкованості
>>> s = 'aa'
>>> t = 'ab'
>>> t > s
True
>>> t + s > s + t
True
```

2. Відношення порядку на послідовностях чисел

```
>>> # Відношення порядку на послідовностях
>>> [1,2] < [1, -3]
False
>>> [1, -3] < [1, 2]
True
>>> [1, 2, -3] < [2, 1, 1]
True
>>> [1.38, 2.01] < [0.0, 5.4]
False
>>> ['piece' , 'peace'] > ['peace', 'piece']
True
```

3. Припускається в одному виразі використання математичних операцій і функцій, булевих операцій та операцій порівняння. При цьому встановлений такий пріоритет операцій: спочатку виконуються математичні обчислення, потім порівняння, а потім булеві операції:

```
>>> import math
>>> 0.5 < math.sin(math.pi/4) and not 2+2 == 5
True
>>>
```

Блоки коду

Незадовбуючи, повторимо і додамо нового..

З точки зору структурного програмування семантичною одиницею програми є блок коду, тобто послідовність інструкцій, кожна з яких, по виконанні, передає управління наступній. Проте, інструкція може бути складною і її виконання полягає у виконанні певної сукупності дій, тобто, внутрішнього блоку коду.

Вказівкою початку й кінця блоку коду для компілятора є так звані операторні дужки. В залежності від синтаксису мови, операторні дужки можуть позначатися по різному. Це зарезервовані слова, наприклад, begin та end у Pascal, фігурні дужки « { } » у C++ та ін.

Оскільки Python підтримує структурне програмування, рядки коду також групуються у блоки. Проте, розробники Python знайшли простий і, водночас, конструктивний спосіб виділення блоків коду.

Продуктивність роботи програміста-кодера багато в чому визначається тим, як він пише, тобто стилем програмування. Існують усталені рекомендації щодо написання коду (наприклад, класична книга Ван-Тассел). У Python основні з цих рекомендацій реалізовані як синтаксичні правила мови Python. Фігулярно кажучи, програміст, використовуючи Python, вимушений притримуватися професійного стилю написання коду.

Зокрема, професійний стиль написання коду передбачає виділення блоків коду відступами. Якщо інструкція передбачає вкладений блок, то рядки блоку записуються з більшим відступом від краю листка, ніж сама інструкція. У Python роль операторних дужок грають відступи.

1. Синтаксичне правило написання коду передбачає, що рядки одного блоку коду повинні мати однакову кількість відступів.
2. Початком вкладеного блоку є рядок який має більшу кількість відступів (як правило, на чотири).
3. Ознакою кінця блоку слугує зменшення відступу наступного рядка до нульової величини або до величини відступу зовнішнього блоку.

Ця особливість синтаксису Python визначна у наступному матеріалі.

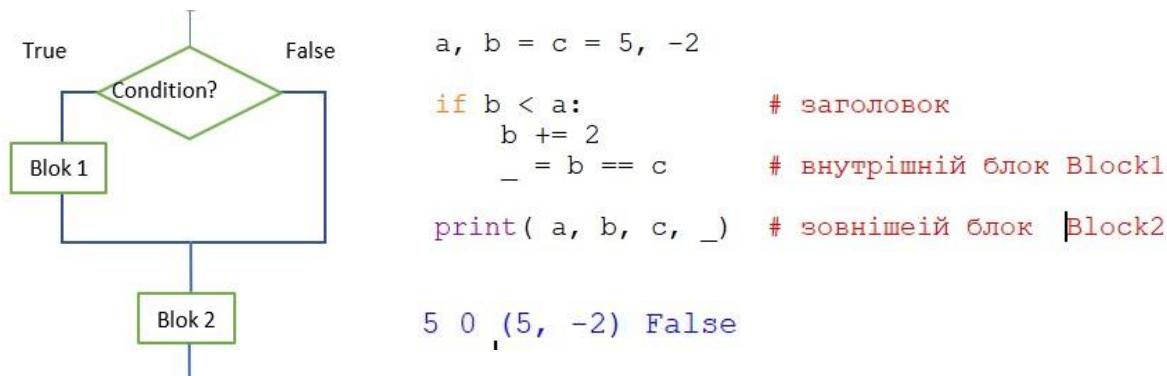
ІНСТРУКЦІЇ гілкування потоку управління

На рисунку вище показані схеми основних алгоритмічних структур управління. У мові Python усі інструкції управління, що відповідають цим алгоритмічним структурам, мають спільний синтаксис – заголовок інструкції, який **обов'язково** закінчується двокрапками, і внутрішній блок коду (тіло інструкції).

Для зручності і спрощення написання інструкцій потоку управління, Python, як і інші мови високого рівня, має декілька варіантів інструкцій для структури гілкування.

Інструкція if

Ця інструкція реалізує алгоритмічну структуру, показану на рисунку



Якщо умова (вираз після **if**) набуває значення True, виконується внутрішній блок Blok1, а потім управління передається наступному за **if** блоку програми, тобто Blok2. Якщо умова набуває значення False, управління одразу передається наступному за **if** блоку програми Blok2.

Інструкція if else Синтаксис і виконання такої інструкції показані на прикладі:

```

Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:/Users/lal02/AppData/Local/Programs/Python/WORK/if_else.py ====
What is your name, please
My name is Alice
Hi, Alice
>>>
==== RESTART: C:/Users/lal02/AppData/Local/Programs/Python/WORK/if_else.py ====
What is your name, please
My name is Kate
Hi, but I dont know u
>>>

```

```

if_else.py - C:/Users/lal02/AppData/Local/Programs/Python/WOR...
File Edit Format Run Options Window Help
print('What is your name, please')
print('My name is ', end=' ')
name = input()
if name == 'Alice':
    print('Hi, ' + name)
else:
    print('Hi, but I don''t know u')

Ln: 8 Col: 0

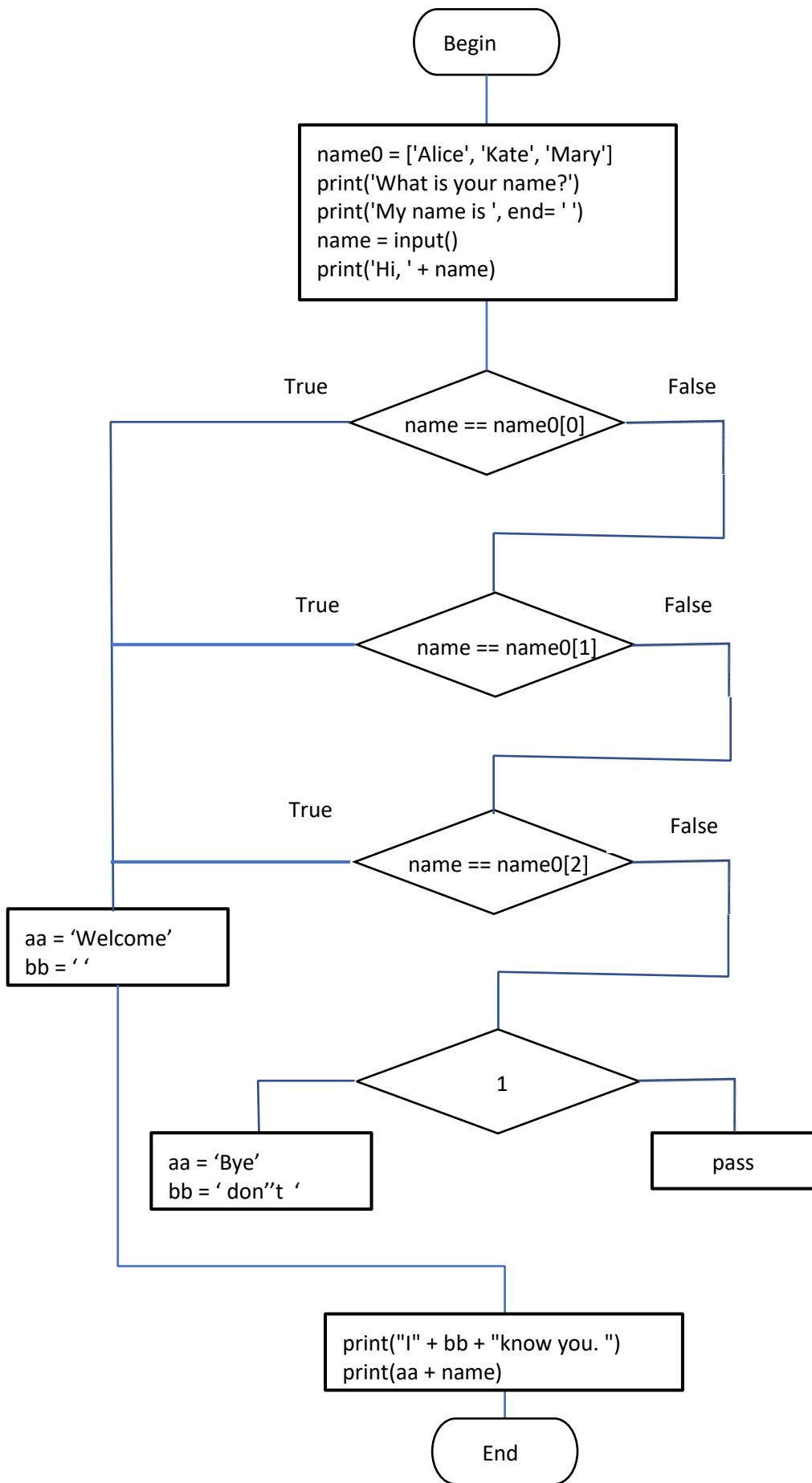
```

Інструкція *elif*

Інструкції **if** та **if else** надають можливість вибору серед двох альтернативних продовжень програми. На практиці дуже часто приходиться вибирати серед більшої кількості альтернатив. Для цього у мові Python (і у деяких інших) використовується інструкція **elif**.

Таких інструкцій може бути вкладено в інструкції **if** стільки, скільки необхідно. Кожна інструкція **elif** містить умову, яка перевіряється лише у тому випадку, коли **всі попередні** умови оказалися хибними. Якщо умова окажеться істиною, виконується внутрішній блок цієї інструкції, а потім управління передається наступному зовнішньому блоку.

Блок схема, синтаксис інструкції **elif** і результати виконання показані на прикладі



```

Python 3.6.3 (v3.6.3:2c5fed8, Oct  3 2017, 17:26:49) [MSC v.
.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more informa
tion.

>>>
===== RESTART: C:\Users\Alex\AppData\Local\Programs\Python\
WORK\el_if.py =====
What is your name?
My name is Bob
Hi, Bob
I don't know you.
Bye, Bob
>>>
===== RESTART: C:\User
WORK\el_if.py =====
What is your name?
My name is Alice
Hi, Alice
I know you.
Welcom, Alice
>>>
===== RESTART: C:\User
WORK\el_if.py =====
What is your name?
My name is Mary
Hi, Mary
I know you.
Welcom, Mary
>>>
===== RESTART: C:\User
WORK\el_if.py =====
What is your name?

```

```

el_if.py - C:\Users\Alex\AppData\Local\Pro...
File Edit Format Run Options Window Help
name0 = ['Alice', 'Kate', 'Mary']
print('What is your name?')
print('My name is ', end=' ')
name = input()
print('Hi, ' + name)

if name == name0[0]:
    aa = 'Welcom, '
    bb = ' '
elif name == name0[1]:
    aa = 'Welcom, '
    bb = ' '
elif name == name0[2]:
    aa = 'Welcom, '
    bb = ' '
else:
    aa = "Bye, "
    bb = " don't "

print("I" + bb + "know you. ")
print(aa + name)

```

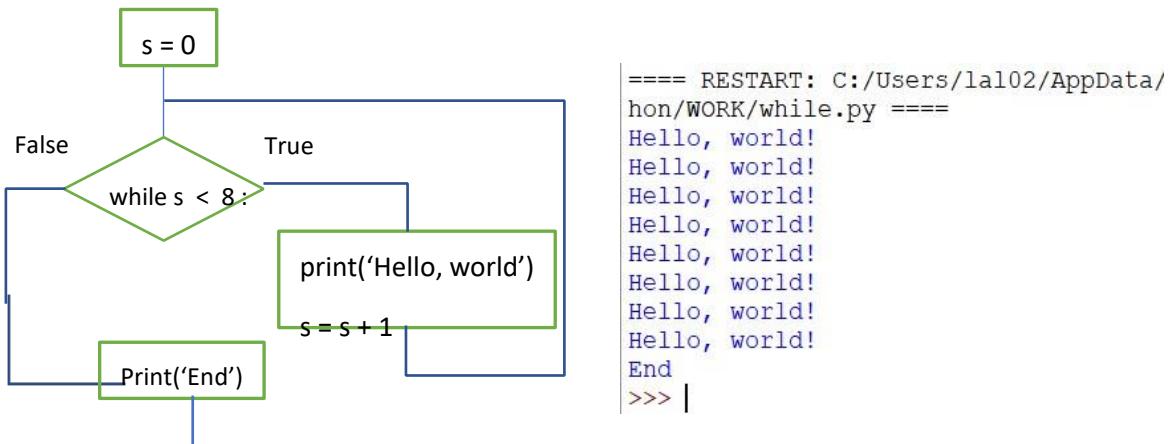
Цікаво, що без інструкції **else** можна обходитись, оскільки вона еквівалентна інструкції **elif** з тотожно істиною умовою. У блок схемі кожному ромбу відповідає умова. Останній містить тотожну умову, що відповідає як інструкції **else**, так і інструкції **elif**.

Інструкції для реалізації циклічних алгоритмів

Цикл while

Інструкція **while** дає змогу організувати многократне повторне виконання внутрішнього блоку коду. Тобто, внутрішній блок коду інструкції **while** виконується до тих пір, поки істина указана у ній умова. Як ні, управління передається наступному зовнішньому блоку коду.

Далі наведений приклад. Для наочності, код вписаний у блоки алгоритмічної структури.



Отже, якщо уявити собі роботу програми, як оброблення певного потоку даних, то інструкція **while** або не втручається у оброблення, якщо дані мають необхідну кондицію, або повернатиме цей потік на повторне оброблення, доки ця кондиція не буде досягнута. У нашому простому прикладі такої кондиції досягає значення змінної `s`.

Нескінчений цикл

Досвідченим програмістам добре відома ситуація так званого «нескінченого циклу», тобто інструкція **while** розпочинає циклічне оброблення даних, але процес не звершується за умовою і продовжується довше припустимого часу.

Як правило, причиною є некоректне формулювання умови виходу із циклу. Нижче наведений простий приклад. Пропонується самостійно виконати цей код, упевнитися, що програма «зациклюється» і встановити причину.

```
*infin_cycle.py - C:/Users/Alex/AppData/Local/Programs/Python/WORK/infin_cycle.py
File Edit Format Run Options Window Help
import math
Pi = math.pi
pp = 0
while pp != Pi:
    print('input constant Pi = ', end= ' ')
    pp = float(input())
print('Ok')
```

У деяких випадках умова може бути сформульована формально правильно, проте несе у собі контекст, неочевидний для користувача. Наступний код-жарт моделює таку ситуацію. Спробуйте, як зациклиться, у чому причина?

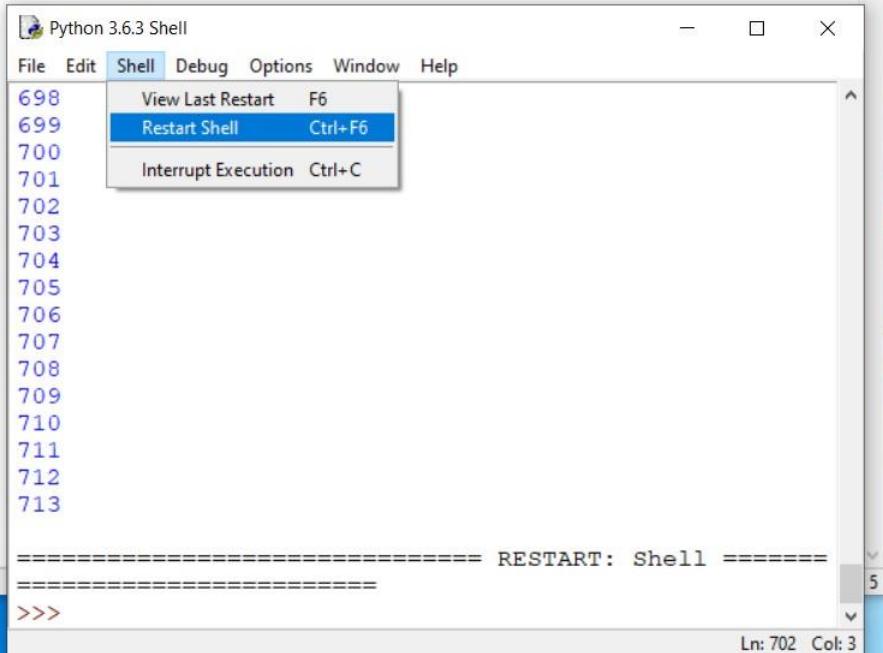
```
*input_name.py - C:/Users/Alex/AppData/Local/Programs/Python/WORK/input_name.py (3.6.3)*
File Edit Format Run Options Window Help
print ('Hi! Type your name =', end=' ')
name = input()
while name != 'your name':
    print ('Don\'t you understood me? Type your name =', end=' ')
    name = input()

print('lastly!!!')
```

Примусове завершення нескінченого циклу

Якщо програма виконується у середовищі IDLE і «зациклилась» із невідомої причини, найрадикальнішими засобами зупинити обчислення є два: старий, ще DOSівський набір клавіш «Ctrl + C», або більш «елегантне» перезавантаження оболонки IDLE. Що показано на рисунку.

На цьому ж рисунку показаний код, який містить більш прихованій контекст, що може привести до нескінченого циклу, і кількість повторів, яка нескінченно зростає. Робота програми була завершена описаним вище способом. У коді **pi** і **Pi** – загальновідоме число Піфагора. **Який контекст?**



```
infin_cycle1.py - C:/Users/Alex/AppData/Local/Programs/Python/WORK/infin_cycle1.py (3.6.3)
File Edit Format Run Options Window Help
import math
Pi = math.pi

print('input costant pi = ', end=' ')
pi = float(input())
print(Pi)
print(pi)
a = 0

while pi != Pi:
    a = a+1
    print(a)

print('Ok')

===== RESTART: Shell =====
>>>
```

Разом з цим, Python містить засоби, які дають змогу програмувати умови санкціонованого оброблення ситуації із зациклуванням програм.

Інструкція break

Існує швидкий засіб примусового виходу програми із циклу. Якщо внутрішній блок інструкції **while** містить інструкцію **break** і алгоритм передає їй управління, то виконання циклу негайно припиняється. Управління передається блоку, наступному за циклом **while**.

Приклад:

Доповнимо внутрішній блок циклу на рисунку вище інструкцією примусового припинення циклу. Умовою переривання буде попередження про некоректність введеного значення константи по відношенню до її машинного значення.

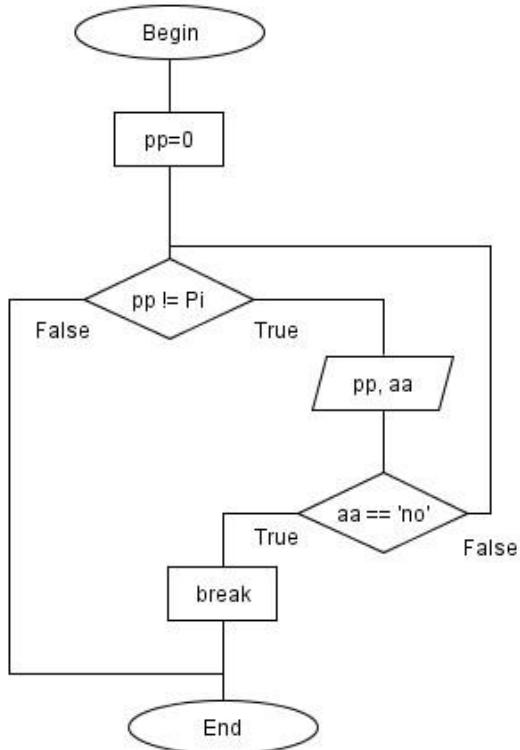
The screenshot shows the Python 3.6.3 IDE interface. The top window is titled 'infin_break.py - C:\Users\Alex\AppData\Local\Programs\Python\WORK\infin_break.py (3.6.3)'. It contains the following code:

```
import math
Pi = math.pi
pp = 0
while pp != Pi:
    print('input const pi = ', end=' ')
    pp = float(input())
    print('Are you sure to continue? Type yes/no', end=' ')
    aa = input()
    if aa == 'no':
        break
print('Ok')
```

The bottom window is titled 'Python 3.6.3 Shell'. It shows the execution of the script:

```
Python 3.6.3 (v3.6.3:2c5fed8, Oct  3 2017, 17:26:49) [MSC v. 1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
== RESTART: C:\Users\Alex\AppData\Local\Programs\Python\WORK\infin_break.py ==
input const pi =  3.14
Are you sure to continue? Type yes/no yes
input const pi =  3.14
Are you sure to continue? Type yes/no no
Ok
````
```

Output coordinates: The main window is at approximately [98, 324, 886, 689]. The shell window is a smaller window within the main interface, located below the main window's title bar.

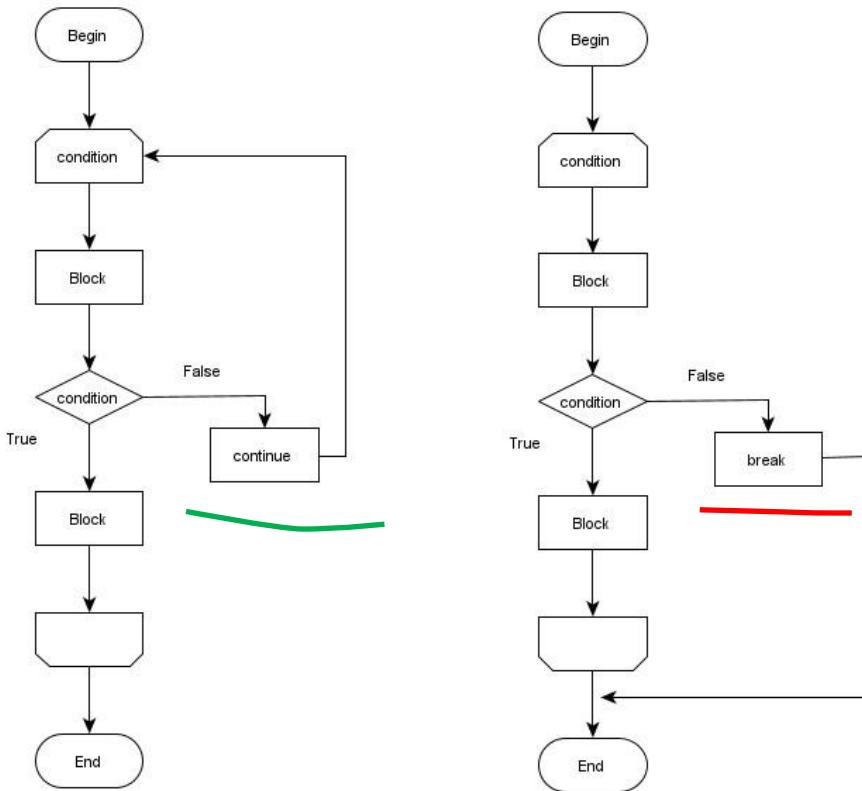


Скритий контекст, що веде до нескінченого циклу такий.

Перші два рядки – завантаження математичної бібліотеки й ініціалізація машинного значення константи  $\text{Pi} = 3.141592653589793$ . Алгоритм входу у нескінчений цикл розрахований на те, що переважна кількість користувачів пам’ятає тільки «шкільне» значення цієї константи **3.14**. Дехто пам’ятає 3.1415. Зрозуміло.

### Інструкція continue

Як й інструкція **break**, інструкція **continue** використовується у циклах. Роль цієї інструкції така сама, як **break** – керувати виконанням циклу: якщо управління передається цим інструкціям, то виконання внутрішнього блоку цикли припиняється. Проте, різниця принципова. **Break** припиняє виконання циклу і передає управління зовнішнім блокам. **Continue** негайно передає управління у початок циклу, де умова обчислюється заново, цикл не переривається.



## Перелічена змінна.

Змінна, всі значення якої можна пронумерувати у математиці називають **“переліченою”**.

Для програміста більш природним буде еквівалентне означення:

Змінна називається **переліченою**, якщо існує деякий алгоритм генерування всіх її значень.

У **Python** перелічена змінна може бути визначена двома способами:

- явно, переліком значень у вигляді списку, словника, масиву тощо;
- алгоритмом, який, у свою чергу, може бути реалізований деякою функцією.

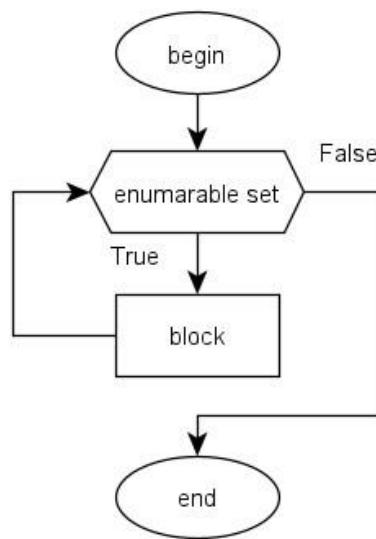
Кожен із способів розглядається далі у прикладах при реалізації циклів.

## Цикл for each

Цикл **do while** виконується до тих пір, поки умова залишається істиною.

Дуже поширеним є частинний випадок, коли умовою циклу є вимога виконати внутрішній блок для **кожного** значення деякої переліченої змінної. У програмуванні таку змінну називають «лічильником циклу» (**counter**).

У силу важливості, такий різновид циклу має власну назву – цикл **for each**. Позначення циклу у блок-схемах показано на рисунку.



Інструкції **for each** у Python мають вигляд:

**for <counter> in <enumerable set>: <Block>**

Як виконується інструкція **for each** вважаємо зрозумілим із рисунку.

### Визначення множини значень лічильника

Лічильник може приймати значення будь-якого типу, що підтримуються у **Python**.

#### Приклад:

Перелічена змінна задається списком простих чисел і задана **явно**, оскільки, як відомо, не існує загального алгоритму генерування простих чисел (до цього питання ще повернемося). На виході – список квадратів цих чисел

```
>>> for i in [11, 13, 17, 19, 23, 31]:
 print(i**2, end= ' ')
```

```
121 169 289 361 529 961
```

#### Приклад:

Рядок у Python, крім прямого призначення, як текст, сприймається і як множина значень переліченої змінної символного типу

```
>>> s = 'ABBA'
>>> 2*s[2:4]
'BABA'
```

Це можна використати для реалізації циклу **for each**:

```
>>> for i in 'Apollo':
 print(ord(i), end=' ')

65 112 111 108 108 111
>>>
```

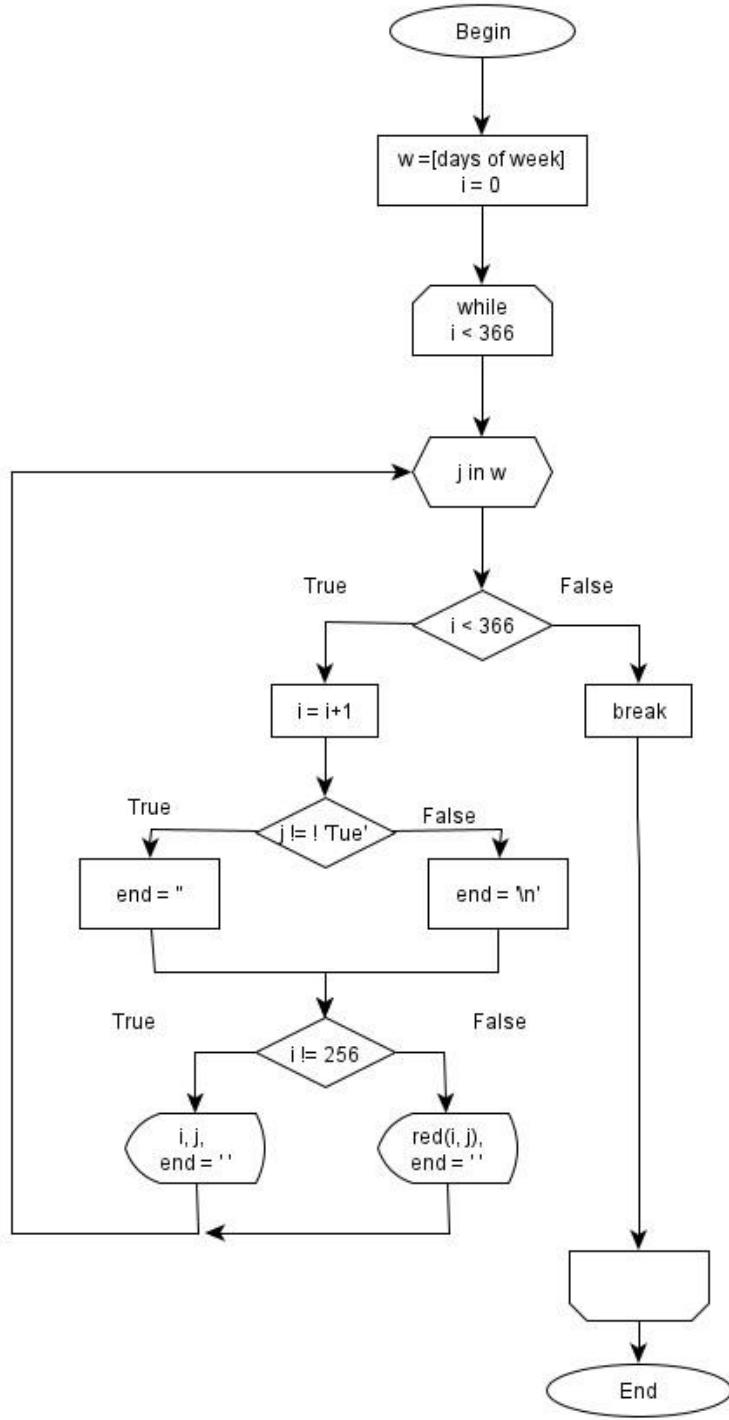
Функція **ord()** повертає Unicode символів рядка.

### *Приклад:*

Наступний приклад, крім циклів, демонструє деякі спроможності **Python** для організації форматного виводу результатів.

Задач програми - встановити відповідність між номером дня 2020-го року і днями тижня, сформувати таблицю і вивести на консоль.

На рисунку наведена блок-схема одного з можливих алгоритмів розв'язування цієї задачі.



Алгоритм полягає у наступному:

1. Ініціалізація списку *w* назв днів тижню, що починається із рядка 'Wed' і закінчується 'Tue', оскільки 1 січня 2020 р. «середа».
2. Ініціалізація лічильника *i*, що необхідно для початку циклу **while do**.
3. Перевірка умови, що значення лічильника *i* менше за кількість днів у 2020 році:
  - 3.1. Якщо **ні**, виконується інструкція **End**.
  - 3.2. Якщо **так**, виконується цикл **for each j in w** – перевіряється умова, значення лічильника *i* < 366 (чому це робиться?):
    - 3.2.1. Якщо **ні**, управління передається циклу **while do**.

**3.2.2. Якщо так:**

3.2.2.1. Інкремен лічильника **i +=1**.

3.2.2.2. Перевірка умови, що лічильник **j** не рівний останньому елементу списку **w** (для чого це робиться?):

3.2.2.2.1. Якщо **так**, виведення на екран здійснюватиметься у поточний рядок.

3.2.2.2.2. Якщо **ні**, виведення на екран здійснюватиметься у новий рядок.

3.2.2.3. Перевірка умови, що значення лічильника **i** не рівне 256:

3.2.2.3.1. Якщо **так**, на екран виводиться рядок блакитного кольору.

3.2.2.3.2. Якщо **ні**, колір рядка на екрані буде червоним.

3.2.2.4. Управління передається на крок 3.2.1

### **Фарбування та форматування тексту.**

Якість дизайну виводу даних – одна із основних характеристик сучасних прикладних програм. Не останнє місце при розробленні дизайну займає кольорове оформлення та форматування тексту.

Базовий набір **Python** не містить графічних функцій. Проте, цей інструментарій добре розвинутий і його можна встановити на комп’ютер за допомогою менеджера **pip**. У цьому прикладі демонструються деякі можливості двох бібліотек. Більш докладно познайомитися з ними можна за наведеними посиланнями.

Форматування тексту при виведенні також можна здійснювати різними функціями. Тут використовується один з найбільш поширених інструментаріїв (<https://pyformat.info/>).

1. Бібліотека **idlecolors** (<https://github.com/lawsie/idlecolors>). Має досить скромні можливості, проте дає змогу здійснювати вивід результату безпосередньо у вікно середовища **IDLE**.

Код, що реалізує наведений алгоритм.

```
2020, 366 days in year
from idlecolors import *
w = ['Wed', 'Thu', 'Fri', 'Sat', 'Sun', 'Mon', 'Tue']
i = 0
|
while i < 366:
 for j in w:
 if i < 366:

 i = i+1

 if j != 'Tue':
 ss = ''
 else:
 ss = '\n'

 if i!=256:
 print('%10s%5s' % (str(i),j), end= ss)
 else:
 printc('%20s' % (red(str(i))) + '%15s' %(red(j)), end = ss)

 else:
 break
```

Результати показані нижче.

```

>>>
= RESTART: C:\Users\lal02\AppData\Local\Programs\Python\Work\for_each_idlecolors1.py
 1 Wed 2 Thu 3 Fri 4 Sat 5 Sun 6 Mon 7 Tue
 8 Wed 9 Thu 10 Fri 11 Sat 12 Sun 13 Mon 14 Tue
 15 Wed 16 Thu 17 Fri 18 Sat 19 Sun 20 Mon 21 Tue
 22 Wed 23 Thu 24 Fri 25 Sat 26 Sun 27 Mon 28 Tue
 29 Wed 30 Thu 31 Fri 32 Sat 33 Sun 34 Mon 35 Tue
 36 Wed 37 Thu 38 Fri 39 Sat 40 Sun 41 Mon 42 Tue
 43 Wed 44 Thu 45 Fri 46 Sat 47 Sun 48 Mon 49 Tue
 50 Wed 51 Thu 52 Fri 53 Sat 54 Sun 55 Mon 56 Tue
 57 Wed 58 Thu 59 Fri 60 Sat 61 Sun 62 Mon 63 Tue
 64 Wed 65 Thu 66 Fri 67 Sat 68 Sun 69 Mon 70 Tue
 71 Wed 72 Thu 73 Fri 74 Sat 75 Sun 76 Mon 77 Tue
 78 Wed 79 Thu 80 Fri 81 Sat 82 Sun 83 Mon 84 Tue
 85 Wed 86 Thu 87 Fri 88 Sat 89 Sun 90 Mon 91 Tue
 92 Wed 93 Thu 94 Fri 95 Sat 96 Sun 97 Mon 98 Tue
 99 Wed 100 Thu 101 Fri 102 Sat 103 Sun 104 Mon 105 Tue
 106 Wed 107 Thu 108 Fri 109 Sat 110 Sun 111 Mon 112 Tue
 113 Wed 114 Thu 115 Fri 116 Sat 117 Sun 118 Mon 119 Tue
 120 Wed 121 Thu 122 Fri 123 Sat 124 Sun 125 Mon 126 Tue
 127 Wed 128 Thu 129 Fri 130 Sat 131 Sun 132 Mon 133 Tue
 134 Wed 135 Thu 136 Fri 137 Sat 138 Sun 139 Mon 140 Tue
 141 Wed 142 Thu 143 Fri 144 Sat 145 Sun 146 Mon 147 Tue
 148 Wed 149 Thu 150 Fri 151 Sat 152 Sun 153 Mon 154 Tue
 155 Wed 156 Thu 157 Fri 158 Sat 159 Sun 160 Mon 161 Tue
 162 Wed 163 Thu 164 Fri 165 Sat 166 Sun 167 Mon 168 Tue
 169 Wed 170 Thu 171 Fri 172 Sat 173 Sun 174 Mon 175 Tue
 176 Wed 177 Thu 178 Fri 179 Sat 180 Sun 181 Mon 182 Tue
 183 Wed 184 Thu 185 Fri 186 Sat 187 Sun 188 Mon 189 Tue
 190 Wed 191 Thu 192 Fri 193 Sat 194 Sun 195 Mon 196 Tue
 197 Wed 198 Thu 199 Fri 200 Sat 201 Sun 202 Mon 203 Tue
 204 Wed 205 Thu 206 Fri 207 Sat 208 Sun 209 Mon 210 Tue
 211 Wed 212 Thu 213 Fri 214 Sat 215 Sun 216 Mon 217 Tue
 218 Wed 219 Thu 220 Fri 221 Sat 222 Sun 223 Mon 224 Tue
 225 Wed 226 Thu 227 Fri 228 Sat 229 Sun 230 Mon 231 Tue
 232 Wed 233 Thu 234 Fri 235 Sat 236 Sun 237 Mon 238 Tue
 239 Wed 240 Thu 241 Fri 242 Sat 243 Sun 244 Mon 245 Tue
 246 Wed 247 Thu 248 Fri 249 Sat 250 Sun 251 Mon 252 Tue
 253 Wed 254 Thu 255 Fri 256 Sat 257 Sun 258 Mon 259 Tue
 260 Wed 261 Thu 262 Fri 263 Sat 264 Sun 265 Mon 266 Tue
 267 Wed 268 Thu 269 Fri 270 Sat 271 Sun 272 Mon 273 Tue
 274 Wed 275 Thu 276 Fri 277 Sat 278 Sun 279 Mon 280 Tue
 281 Wed 282 Thu 283 Fri 284 Sat 285 Sun 286 Mon 287 Tue
 288 Wed 289 Thu 290 Fri 291 Sat 292 Sun 293 Mon 294 Tue
 295 Wed 296 Thu 297 Fri 298 Sat 299 Sun 300 Mon 301 Tue
 302 Wed 303 Thu 304 Fri 305 Sat 306 Sun 307 Mon 308 Tue
 309 Wed 310 Thu 311 Fri 312 Sat 313 Sun 314 Mon 315 Tue
 316 Wed 317 Thu 318 Fri 319 Sat 320 Sun 321 Mon 322 Tue
 323 Wed 324 Thu 325 Fri 326 Sat 327 Sun 328 Mon 329 Tue
 330 Wed 331 Thu 332 Fri 333 Sat 334 Sun 335 Mon 336 Tue
 337 Wed 338 Thu 339 Fri 340 Sat 341 Sun 342 Mon 343 Tue
 344 Wed 345 Thu 346 Fri 347 Sat 348 Sun 349 Mon 350 Tue
 351 Wed 352 Thu 353 Fri 354 Sat 355 Sun 356 Mon 357 Tue
 358 Wed 359 Thu 360 Fri 361 Sat 362 Sun 363 Mon 364 Tue
 365 Wed 366 Thu

```

>>> |

2. Бібліотека **colorama** (<https://pypi.org/project/colorama/>) має широкий спектр можливостей. Вона орієнтована на професійних програмістів. Виведення результатів здійснюється у командний рядок терміналу операційної системи (як у нашому прикладі), або у окреме вікно, що є складовою інтерфейсу виконуваної програми. Код (рис. ) інтуїтивно зрозумілий. Додамо тільки, що інструкція

## print (Back.WHITE)

змінює чорний фон виведення на білий колір.

Код і результати показані на рисунках

 for\_each - Notepad

File Edit Format View Help

# 2020, 366 days in year

```
import colorama
from colorama import Fore, Back, Style
colorama.init()

w = ['Wed', 'Thu', 'Fri', 'Sat', 'Sun', 'Mon', 'Tue']
i = 0
print(Back.WHITE)
while i < 366:
 for j in w:
 if i < 366:

 i = i+1

 if j != 'Tue':
 ss = '':
 else:
 ss = '\n'

 if j != 256:
 print(Fore.BLUE + '%10d%5s' % (i,j),end= ss)
 else:
 print(Fore.RED + '%10d%5s' % (i,j), end= ss)

 else:
 break
```

**Визначення переліченої змінної функцією (алгоритмом).**

```
C:\Users\la102>python for_each.py
```

|     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | Wed | 2   | Thu | 3   | Fri | 4   | Sat | 5   | Sun | 6   | Mon | 7   | Tue |
| 8   | Wed | 9   | Thu | 10  | Fri | 11  | Sat | 12  | Sun | 13  | Mon | 14  | Tue |
| 15  | Wed | 16  | Thu | 17  | Fri | 18  | Sat | 19  | Sun | 20  | Mon | 21  | Tue |
| 22  | Wed | 23  | Thu | 24  | Fri | 25  | Sat | 26  | Sun | 27  | Mon | 28  | Tue |
| 29  | Wed | 30  | Thu | 31  | Fri | 32  | Sat | 33  | Sun | 34  | Mon | 35  | Tue |
| 36  | Wed | 37  | Thu | 38  | Fri | 39  | Sat | 40  | Sun | 41  | Mon | 42  | Tue |
| 43  | Wed | 44  | Thu | 45  | Fri | 46  | Sat | 47  | Sun | 48  | Mon | 49  | Tue |
| 50  | Wed | 51  | Thu | 52  | Fri | 53  | Sat | 54  | Sun | 55  | Mon | 56  | Tue |
| 57  | Wed | 58  | Thu | 59  | Fri | 60  | Sat | 61  | Sun | 62  | Mon | 63  | Tue |
| 64  | Wed | 65  | Thu | 66  | Fri | 67  | Sat | 68  | Sun | 69  | Mon | 70  | Tue |
| 71  | Wed | 72  | Thu | 73  | Fri | 74  | Sat | 75  | Sun | 76  | Mon | 77  | Tue |
| 78  | Wed | 79  | Thu | 80  | Fri | 81  | Sat | 82  | Sun | 83  | Mon | 84  | Tue |
| 85  | Wed | 86  | Thu | 87  | Fri | 88  | Sat | 89  | Sun | 90  | Mon | 91  | Tue |
| 92  | Wed | 93  | Thu | 94  | Fri | 95  | Sat | 96  | Sun | 97  | Mon | 98  | Tue |
| 99  | Wed | 100 | Thu | 101 | Fri | 102 | Sat | 103 | Sun | 104 | Mon | 105 | Tue |
| 106 | Wed | 107 | Thu | 108 | Fri | 109 | Sat | 110 | Sun | 111 | Mon | 112 | Tue |
| 113 | Wed | 114 | Thu | 115 | Fri | 116 | Sat | 117 | Sun | 118 | Mon | 119 | Tue |
| 120 | Wed | 121 | Thu | 122 | Fri | 123 | Sat | 124 | Sun | 125 | Mon | 126 | Tue |
| 127 | Wed | 128 | Thu | 129 | Fri | 130 | Sat | 131 | Sun | 132 | Mon | 133 | Tue |
| 134 | Wed | 135 | Thu | 136 | Fri | 137 | Sat | 138 | Sun | 139 | Mon | 140 | Tue |
| 141 | Wed | 142 | Thu | 143 | Fri | 144 | Sat | 145 | Sun | 146 | Mon | 147 | Tue |
| 148 | Wed | 149 | Thu | 150 | Fri | 151 | Sat | 152 | Sun | 153 | Mon | 154 | Tue |
| 155 | Wed | 156 | Thu | 157 | Fri | 158 | Sat | 159 | Sun | 160 | Mon | 161 | Tue |
| 162 | Wed | 163 | Thu | 164 | Fri | 165 | Sat | 166 | Sun | 167 | Mon | 168 | Tue |
| 169 | Wed | 170 | Thu | 171 | Fri | 172 | Sat | 173 | Sun | 174 | Mon | 175 | Tue |
| 176 | Wed | 177 | Thu | 178 | Fri | 179 | Sat | 180 | Sun | 181 | Mon | 182 | Tue |
| 183 | Wed | 184 | Thu | 185 | Fri | 186 | Sat | 187 | Sun | 188 | Mon | 189 | Tue |
| 190 | Wed | 191 | Thu | 192 | Fri | 193 | Sat | 194 | Sun | 195 | Mon | 196 | Tue |
| 197 | Wed | 198 | Thu | 199 | Fri | 200 | Sat | 201 | Sun | 202 | Mon | 203 | Tue |
| 204 | Wed | 205 | Thu | 206 | Fri | 207 | Sat | 208 | Sun | 209 | Mon | 210 | Tue |
| 211 | Wed | 212 | Thu | 213 | Fri | 214 | Sat | 215 | Sun | 216 | Mon | 217 | Tue |
| 218 | Wed | 219 | Thu | 220 | Fri | 221 | Sat | 222 | Sun | 223 | Mon | 224 | Tue |
| 225 | Wed | 226 | Thu | 227 | Fri | 228 | Sat | 229 | Sun | 230 | Mon | 231 | Tue |
| 232 | Wed | 233 | Thu | 234 | Fri | 235 | Sat | 236 | Sun | 237 | Mon | 238 | Tue |
| 239 | Wed | 240 | Thu | 241 | Fri | 242 | Sat | 243 | Sun | 244 | Mon | 245 | Tue |
| 246 | Wed | 247 | Thu | 248 | Fri | 249 | Sat | 250 | Sun | 251 | Mon | 252 | Tue |
| 253 | Wed | 254 | Thu | 255 | Fri | 256 | Sat | 257 | Sun | 258 | Mon | 259 | Tue |
| 260 | Wed | 261 | Thu | 262 | Fri | 263 | Sat | 264 | Sun | 265 | Mon | 266 | Tue |
| 267 | Wed | 268 | Thu | 269 | Fri | 270 | Sat | 271 | Sun | 272 | Mon | 273 | Tue |
| 274 | Wed | 275 | Thu | 276 | Fri | 277 | Sat | 278 | Sun | 279 | Mon | 280 | Tue |
| 281 | Wed | 282 | Thu | 283 | Fri | 284 | Sat | 285 | Sun | 286 | Mon | 287 | Tue |
| 288 | Wed | 289 | Thu | 290 | Fri | 291 | Sat | 292 | Sun | 293 | Mon | 294 | Tue |
| 295 | Wed | 296 | Thu | 297 | Fri | 298 | Sat | 299 | Sun | 300 | Mon | 301 | Tue |
| 302 | Wed | 303 | Thu | 304 | Fri | 305 | Sat | 306 | Sun | 307 | Mon | 308 | Tue |
| 309 | Wed | 310 | Thu | 311 | Fri | 312 | Sat | 313 | Sun | 314 | Mon | 315 | Tue |
| 316 | Wed | 317 | Thu | 318 | Fri | 319 | Sat | 320 | Sun | 321 | Mon | 322 | Tue |
| 323 | Wed | 324 | Thu | 325 | Fri | 326 | Sat | 327 | Sun | 328 | Mon | 329 | Tue |
| 330 | Wed | 331 | Thu | 332 | Fri | 333 | Sat | 334 | Sun | 335 | Mon | 336 | Tue |
| 337 | Wed | 338 | Thu | 339 | Fri | 340 | Sat | 341 | Sun | 342 | Mon | 343 | Tue |
| 344 | Wed | 345 | Thu | 346 | Fri | 347 | Sat | 348 | Sun | 349 | Mon | 350 | Tue |
| 351 | Wed | 352 | Thu | 353 | Fri | 354 | Sat | 355 | Sun | 356 | Mon | 357 | Tue |
| 358 | Wed | 359 | Thu | 360 | Fri | 361 | Sat | 362 | Sun | 363 | Mon | 364 | Tue |
| 365 | Wed | 366 | Thu |     |     |     |     |     |     |     |     |     |     |

1. **Функція користувача.** Інструкція `for each` припускає формування значень переліченої змінної за допомогою функції користувача. У прикладі користувач визначає функцію  $f(x)$ , яка потім використовується для організації циклу:

```
>>> def f(x):
 if x > 0:
 return ['1', '3']
 else:
 return ['2', '4']
```

```
>>> for i in f(3):
 print('a' + i)
```

```
a1
a3
>>> for i in f(-1):
 print('a' + i)
```

```
a2
a4
>>> for i in f(True):
 print('a' + i)
```

```
a1
a3
>>> for i in f(False):
 print('a' + i)
```

```
a2
a4
>>>
>>> int(True)
1
>>> int(False)
0
```

2. **Функція range()**. Випадок, коли лічильник приймає числові значення, настільки поширений, що у Python для цього є спеціальна функція **range()**, яка генерує дані окремого типу ‘**range**’ – множину рівновіддалених між собою **цілих** чисел в межах заданого інтервалу

```
>>> r = range(-2, 10, 3)
>>> type(r)
<class 'range'>
>>>
```

<https://matplotlib.org/genindex.html>

[https://matplotlib.org/3.1.0/api/\\_as\\_gen/matplotlib.pyplot.scatter.html](https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.scatter.html)

<http://python-simple.com/python-matplotlib/scatterplot.php>

[https://matplotlib.org/3.1.1/api/\\_as\\_gen/matplotlib.pyplot.subplot.html](https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.subplot.html)

Загальний вигляд інструкції містить три параметри цілого типу:

### range(start, stop, step)

**start** – ліва межа інтервалу **stop** – права межа інтервалу **step** – крок (відстань між значеннями лічильника на інтервалі)

На рисунку показаний код і результати його виконання.

```
range.py - C:/Users/lal02/OneDrive/Cathedra/2018
File Edit Format Run Options Window Help
print('range(3, 25, 4)')
for count in range(3, 25, 4):
 print(count)
print(end= '\n')

print('range(3, 25, 6)')
for count in range(3, 25, 6):
 print(count)
```

```
>>>
= RESTART: C:/Users/lal02/OneDrive/2019/Auto/range.py
range(3, 25, 4)
3
7
11
15
19
23

range(3, 25, 6)
3
9
15
21
>>>
```

```
>>> type(range(10))
<class 'range'>
>>> type(range(10)[5])
<class 'int'>
>>> |
```

**Зауважимо**, що кількість ітерацій циклу визначається не значенням лічильника, а **кількістю** значень лічильника на заданому інтервалі, що видно на рисунку.

За замовчанням, параметри **start** й **step** мають значення, відповідно 0 і 1. При використання значень за замовчанням можна використовувати спрощені вирази для функції. За результатом виконання вирази однакові

**range(start, stop, 1)** і **range(start, stop)**  
**range(0, stop, 1)** і **range(stop)**

Зауважимо, що результатом виконання **range(stop)** будуть номери 0, 1, 2, ....stop

```
>>> for j in range(2, 10, 1):
 print(j, end= ' ')
```

2 3 4 5 6 7 8 9

```
>>> for j in range(2, 10):
 print(j, end= ' ')
```

2 3 4 5 6 7 8 9

```
>>> for j in range(0, 12, 1):
 print(j, end= ' ')
```

0 1 2 3 4 5 6 7 8 9 10 11

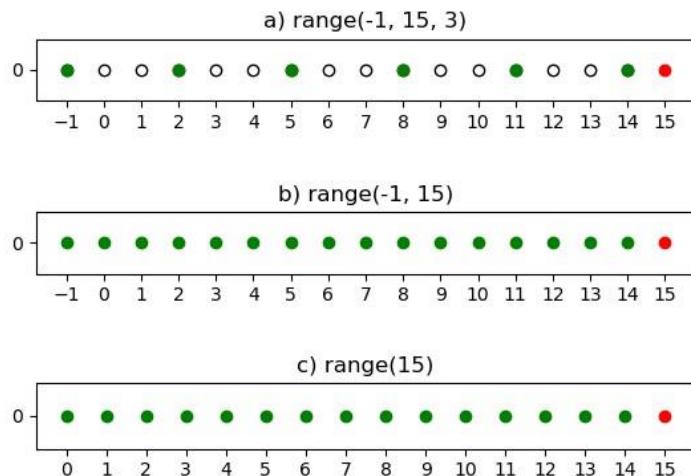
```
>>> for j in range(10):
 print(j, end= ' ')
```

0 1 2 3 4 5 6 7 8 9

```
>>> |
```

Нижче на рисунку точки візуалізують множину значень згенерованих функцією при різних значеннях параметрів.

### Example possible formats function range()



### Функція range ( ) з від'ємним кроком (декрементація)

Якщо параметр **step** додатній, то **range (start, stop, step)** генерує зростаючу послідовність цілих чисел (**інкрементація**). Функція **range (start, stop, step)** припускає і від'ємне значення параметру **step** і ми отримаємо спадаючу послідовність цілих чисел (**декрементація**).

У наступному прикладі **step = -2**. Це означає, що декрементація буде рівна 2 для кожного циклу:

```
>>> for j in range(10, -6, -3):
 print(j, end=' | ')
```

```
10 | 7 | 4 | 1 | -2 | -5
```

Особливістю функції **range()** є те, що вона сприймає рядки як об'єкт типу '**range**'.

Цікавою можливістю **Python** є функція **reversed()**, яка змінює упорядкованість списку на протилежну. У наведеному нижче прикладі формуються пари значень реверсивних списків

```
>>> s = 'abc'
>>> for i in reversed(s):
 for j in reversed(range(5)):
 print([i, j])

['c', 4]
['c', 3]
['c', 2]
['c', 1]
['c', 0]
['b', 4]
['b', 3]
['b', 2]
['b', 1]
['b', 0]
['a', 4]
['a', 3]
['a', 2]
['a', 1]
['a', 0]
>>>
```

**Звертаємо увагу, що `reversed()` і цикл `for each` у Python працюють з рядками, як переліченою змінною.**

### Ще декілька деталей:

1. Результатом роботи `range()` є об'єкт спеціального типу 'range', з яким працюють далеко не всі базові функції, зокрема `print()`. Проте, структурні частини цього об'єкту досяжні, як у звичайної змінної з індексом (список, словник і т.і.) (рис.)

```
>>> s = range(15, -2, -3)
>>> type(s)
<class 'range'>
>>> print(s[2])
9
```

2. Якщо продовжити експеримент з `range()`, як із індексною змінною, то отримаємо ще одну можливість Python

```
>>> s[2:4]
range(9, 3, -3)
>>> for j in s[2:4]:
 print(j)
```

```
9
6
>>>
```

Перший рядок коду звичайно означає, що вибираються значення індексної змінної з номерами 2 та 3. Проте, у **Python** результат виконання цього коду дещо

неподіваний – згенерована функція `range(9, 3, -3)`, яка є «зрізом» вихідного об'єкту і далі використовується у циклі.

Для тих, хто цікавиться архітектурою даних **Python** нижче проведений експеримент. Повернемося до попереднього коду і дослідимо архітектуру даних. Створено об'єкт `s` типу ‘`range`’ і показано його вміст. Далі визначається його адреса у RAM.

```
>>> s = range(15, -2, -3)
>>> for j in s:
 print(j)
```

```
15
12
9
6
3
0
>>> id(s)
1860876958960
```

Далі формується об'єкт `t` типу ‘`range`’ і показано його вміст

```
>>> t = s[2:4]
>>> type(t)
<class 'range'>
>>> print(t)
range(9, 3, -3)
>>> for j in t:
 print(j)
```

```
9
6
>>>
```

## Float i range()

Об'єкт `range()` утворений цілими числами. Якщо викликати `range()` з десятковим числом, побачимо повідомлення про помилку:

```
>>> type(range(5))
<class 'range'>
>>> type(range(5.2))
Traceback (most recent call last):
 File "<pyshell#3>", line 1, in <module>
 type(range(5.2))
TypeError: 'float' object cannot be interpreted as an integer
```

Разом з цим, існує безліч задач, у яких лічильник повинен приймати дійсні значення. Наприклад, при використанні чисельних методів.

Для цієї мети можна використати метод `arange()` із бібліотеки `numpy`. Ця бібліотека може бути встановлена за допомогою менеджера `pip` (див. вище). Синтаксис виразу і змістъ параметрів методу цілком аналогічний `range()`

```
>>> import numpy as np
>>> t = np.arange(0.3, 1.6, 0.2)
 arange([start,] stop[, step,], dtype=None)
```

Тому обмежимся простим прикладом:

```
>>> t = np.arange(0.3, 1.6, 0.2)
>>> type(t)
<class 'numpy.ndarray'>
>>> t
array([0.3, 0.5, 0.7, 0.9, 1.1, 1.3, 1.5])
>>>
```

На відзнаку від `range()`, яка генерує дані спеціального типу, метод `arange()` *генерує* масив.

### **Імпортування модулів** (Відомо, повторення мати навчання).

Будь-якій програмі мовою **Python** досяжний базовий набір функцій, які називаються вбудованими, наприклад, такі: `print()`, `input()`, `len()` та ін., з якими вже встигли познайомитися. Крім того, у поставку Python також входить набір функцій, який називається *стандартною бібліотекою*. Для зручності у використанні ці функції за близькістю тематики поділені на іменовані групи. Такі групи будемо називати модулями.

Наприклад, модуль `math` включає математичні функції, модуль `random` – функції для роботи зі випадковими величинами, і т. п.

Оскільки **Python** є відкритим ресурсом, на сайті розробників можна отримати доступ до каталогу модулів, розроблених користувачами **Python**.

**Python** – універсальна мова програмування. Тому недоцільно у сеансі завантажувати увесь цей інструментарій разом з ядром. Тому, цілком природним було оснастити **Python** менеджером `pip`, який забезпечує можливість встановлювати на комп’ютер всі потенційно необхідні модулі, а також реалізувати інструкцію `import`, яка забезпечить інтеграцію ядра з модулями і функціями, необхідними безпосередньо у сеансі роботи з **Python**.

Робота з менеджером `pip` описана вище.

Інструкція `import` складається із наступних елементів:

- ключове слово `import`;
- через пробіл перелік імен модулів (можливо один), які будуть завантажені для використання на протязі сеансу.

Якщо модуль імпортований, можна використовувати будь-яку функцію, що він містить. Як приклад, згенеруємо послідовності цілих і дійсних випадкових

чисел за допомогою функції **randint** ( ), яка входить у модуль **random**, цілочисленного і дійсного лічильників (відповідно функції **range()**, яка входить у базовий набір, і **arange()**, яка входить у модуль **numpy**).

### Приклад:

```
>>> import random, numpy
>>> for i in range(5):
 print(random.randint(1, 10), end=' ')

10 4 4 2 1
>>> for i in numpy.arange(3.1, 9.6, 0.7):
 print(random.randint(1, 10), end=' ')

8 10 4 3 4 7 7 7 10 10
>>>
```

Ім'я функції, що застосовується програмою, відокремлюється крапкою від імені модуля, якому вона належить. Досить часто імена модулів досить довгі. Полегшати написання коду можна, замінивши ці імена більш простими альтернативними. Наведений вище приклад можна замінити еквівалентним

```
>>> import random as rn, numpy as np
>>> for i in range(5):
 print(rn.randint(1, 10), end=' ')

4 5 1 1 4
>>> for i in np.arange(3.1, 9.6, 0.7):
 print(rn.randint(1, 10), end=' ')

8 1 3 3 4 8 4 3 6 3
>>> |
```

Коди на рисунках подібні, проте результати різні. **Чому?**

Інструкція **import** надає можливість використовувати будь-яку функцію імпортованого модуля. Проте, досить часто програміст знає наперед, що буде використовувати лише деякі з них і імпортувати увесь модуль немає сенсу.

Щоб імпортувати тільки необхідні функції можна використати альтернативну форму інструкції **import()**, яка складається із слова **from**, за яким слідує ім'я модуля, потім слово **import** і перелік функцій, які необхідно імпортувати.

Зауважимо, що після виконання такої форми інструкції не має необхідності зліва від імені функції через крапку вказувати ім'я модуля. Порівняйте еквівалентні коди на рисунках.

```
>>> from random import randint
>>> for i in range(5):
 print(randint(1,10), end=' ')
8 9 1 6 7
```

Альтернативна форма інструкції дає змогу імпортувати і всі функції модуля. На рисунку показана інструкція імпортувати всі функції модуля random. Природно, що при використанні такої форм інструкції також немає необхідності зліва від імені функції через кому вказувати імя модуля

```
>>> from random import *
>>> for i in range(5):
 print(randint(1,10), end=' ')
10 9 3 2 5
```

### Примусове завершення програми за допомогою методу sys.exit()

Практика показує на доцільність існування можливості примусово закінчити виконання програми. Наприклад, при налагодженні або тестуванні програми.

Вище розглядалися засоби оболонки IDLE. Проте критично необхідно мати засоби самого Python, які зробили би цю можливість незалежної від середовища, у якому виконується програма.

Подібним засобом є інструкція **break**. Проте, передбачається примусова передача управління інструкції наступній за блоком (наприклад, цикл), якому належить **break**. Тобто, виконання програми продовжується.

Оптимальним є наявність модуля, що забезпечує взаємодію із операційною системою, під управлінням якої працює **Python**, і наявності у цьому модулі функції примусового завершення програми. Таким є модуль **sys** (див. вище), який містить функцію **exit ()**. На прикладі штучно утвореного нескінченого циклу продемонструємо роботу цієї функції

```
>>> from sys import exit
>>> while True:
 print('Type exit to exit')
 response = input()
 if response == 'exit':
 exit()
```

```
Type exit to exit
v
Type exit to exit
exit
```

У коді є нескінчений цикл, у якому відсутня інструкція **break**. Єдина можливість завершити роботу цієї програми – це ввести рядок *exit* у відповідь на запит, що приведе до виклику функції **exit ()**.

## Контрольні питання

1. Які два можливих значення даних булева типу? Як вони записуються?

2. Назвіть три булеві операції.

3. Напишіть таблиці істинності для кожної з булевих операцій. 4.

Обчисліть значення виразів

$(5 > 4) \text{ and } (3 \text{ not } (5 > 4))$   $(5 > 4) \text{ or } (3 \text{ not } ((5 > 4) \text{ or } (3 == 5)))$

$(\text{True and True}) \text{ and } (\text{True} == \text{False}) \text{ or } (\text{not True})$

5. Назвіть шість операцій порівняння.

6. Чим відрізняються операції рівності й присвоювання?

7. Що таке «умова», як вираз мови Python, і де використовується умови?

8. Ідентифікуйте три блоки у наведеному коді.

```
spam = 0
if spam == 10:
 print('eggs')
 if spam > 5:
 print('bacon')
 else:
 print('ham')
 print('spam')
print('spam')|
```

9. Напишіть код, який виводить різні повідомлення в залежності від значення, змінної *spam*: Hello – при значенні 1, Howdy – при значенні 2 і Greetings ! – при будь-якому іншому значенні.

10. Яку комбінацію клавіш слід натиснути, щоб вивести програму із нескінченого циклу?

11. Чим відрізняються інструкції *break* і *continue*?

12. Чим відрізняються виклики функцій *range(10)* , *range(0, 10)* і *range(0, 10, 1)* у циклі *for*?

13. Напишіть коротку програму, яка виводить числа від 1 до 10 за допомогою циклу **for**. Потім напишіть аналогічну програму, у якій використовується цикл **while**.

14. Як викликати деяку функцію *f()* , яка зберігається у модулі *spam*, після того, як імпортували цей модуль?

## Завдання (у порядку зростання складності):

1. Розробити програму, яка буде вгадувати число, яке загадав користувач. Побудувати алгоритм і подати його у вигляді псевдокоду, блок-схеми, лістингу

мовою Python. Результати подати у вигляді скріншоту діалогу користувача і програми.

**Обмеження:** Число, що загадує користувач, із проміжку [0, 100].

**Стратегія:** Програма ділить проміжок навпіл і запитує більше, чи менше середини проміжку загадане число. Отримавши новий проміжок у середині якого знаходиться загадане число, дії повторюються. Стратегія В:

2. Розробити програму, яка пропонує користувачу вгадати число.

Побудувати алгоритм і подати його у вигляді псевдокоду, блок-схеми, лістингу мовою Python. Результати подати у вигляді скріншоту діалогу користувача і програми.

**Обмеження:** Число, що загадує програма, із проміжку [10, 99].

**Стратегія:** Програма генерує випадкове число у вказаному проміжку і зберігає його. Користувач пропонує програмі пару цифр (**не число!**). Програма порівнює цю пару із загаданим числом і повертає користувачу відповідь із переліку можливих:

- число не містить жодної з цифр;
- число містить одну з цифр, але вона стоїть не на своєму місці;
- число містить одну з цифр і вона стоїть на своєму місці;
- число містить обидві цифри, але вони стоять не на своїх місцях;
- число вгадане.

3. Розробити програму, яка буде вгадувати число, загадане користувачем число.

Побудувати алгоритм і подати його у вигляді псевдокоду, блок-схеми, лістингу мовою Python. Результати подати у вигляді скріншоту діалогу користувача і програми.

**Обмеження:** Число, що загадує користувач, із проміжку [10, 99].

**Стратегія:** У цьому завданні користувач і програма міняються місцями по відношенню до п. 2. Користувач загадує число у вказаному проміжку і зберігає його. Програма пропонує користувачу пару цифр (**не число!**). Користувач порівнює цю пару із загаданим числом і повертає програмі відповідь із переліку можливих:

- число не містить жодної з цифр;
- число містить одну з цифр, але вона стоїть не на своєму місці;
- число містить одну з цифр і вона стоїть на своєму місці;
- число містить обидві цифри, але вони стоять не на своїх місцях;
- число вгадане.

**Додаткове завдання.** Пошукайте в Інтернет інформацію про функції round ( ) і abs ( ). З'ясуйте, що вони виконують. Самостійно проекспериментуйте з ними в інтерактивній оболонці

## Лабораторна робота 8

### Тема: «Функції»

#### *Мета:*

1. Поновити поняття функцій.
2. Функції у *Python*, як засіб повторного використання коду у структурному програмуванні.
3. Відповісти на контрольні питання та виконати завдання.

### Теоретичний мінімум Функції

#### і процедури

Часто ототожнюють поняття функції і процедури. З освітнянської точки зору варто саме тут означити різницю між ними.

Процедура і функція є іменованим блоком інструкцій і є втіленням прогресивної технології програмування - повторного використання коду.

Функція **обов'язково** повертає у точку виклика обчислене значення строго означеного типу (тип функції). Найпростішою функцією є присвоювання.

Процедура виконує певні дії, передбачені алгоритмом, проте **ніяких значень** у точку виклика не повертає. Найпростішою процедурою є окремий оператор. У цьому і полягає принципова різниця і за змістом, і за реалізацією між цими поняттями.

Як відомо, у мові C++ та її розвитках, немає процедур, тільки функції. З точки однорідності реалізації це так. Проте, з точки зору алгоритмізації, без процедур не можна обходитися. У C++ введений спеціальній тип **void**. **Показчик** цього типу не вказує ні на один об'єкт у пам'яті комп'ютера. Отже, фактично, процедура у C++ - це функція типу **void**. Ідея однорідної реалізації процедур й функцій настільки продуктивна, що тип даних, подібний до **void**, реалізований у більшості об'єктно-орієнтованих мовах. У Python це тип NoneType.

Наприклад, використана вище функція **print()**, по суті, є процедурою, яка виконує дію - друкує на екрані значення аргументу, і не повертає у програму ніякого значення:

```
>>> print('a')
a
>>> z = print('a') # можна і так
a
>>> print(z) # яке значення повертається?
None
>>> type(z) # який тип змінної z?
<class 'NoneType'>
```

Отже, це процедура.

Проекспериментуємо з інструкцією `input()`. `len()` – функції, оскільки повертають у програму результат своїх дій:

```
>>> # print - процедура, яка виконує дію - вивід рядка на екран
>>> print('a')
a
>>> # можливо, це функція, яка щось повертає у програму
>>> z = print('a')
a
>>> x = input(' Hello, user! ') # пропозиція ввести с консолі дані у програму
Hello, user! Hello, program!
>>> x # змінна вказує на об'єкт, введений у програму
'Hello, program!'
>>> type(x) # тип цих даних
<class 'str'>
```

Аналогічно

```
>>> y = len(x) # довжина цього рядка
>>> y
15
>>> type(y)
<class 'int'>
```

Отже, `len()` – функція, яка повертає у програму дані цілого типу. Виконаємо таку дію:

```
>>> len(x)
15
```

Якщо `len()` функція, то яка ж змінна вказує на значення 15, що повертається у програму у цьому випадку?

У багатьох мовах високого рівня реалізований апарат так званих **анонімних змінних**. У **Python** така змінна за замовчанням має ідентифікатор « `_` ». Саме ця змінна і вказує на результат у попередньому прикладі. Продовжимо його:

```
>>> _
15
>>> type(_)
<class 'int'>
>>> z = id(_)
>>> z
1794946439600
>>> type(z)
<class 'int'>
>>> _
<class 'int'>
```

Змінна «`_`» вказує на ціле число **15**. На адресу цього об'єкту вказує змінна **z** цілого типу.

Анонімні змінні доцільно використовувати тоді, коли результат, що повертається, не передбачається використовувати у подальшому.

## Інструкція def

Для створення функцій користувача використовується інструкція **def**. Інструкція для створення функції має такий формат:

```
def name(<список аргументів>):
 блок інструкцій return
 <список виводу>
```

Ключові слова **def** – означає початок заготовки функції, **return** – список значень, що повертається функцією. Слово, **name** – ім'я користувацької функції.

### *Приклад:*

Функція **h(a, b, c)**, яка розв'язує квадратне рівняння, розроблена і записана як файл **first.py** у кореневу папку Python. При організації виводу результату ця функція використовує функцію **g(a)**, яка також **записана** у файл **first.py**. Таким чином, цей приклад демонструє створення модуля, аналогічно тим, що використовувалися у попередніх прикладах.

```

def g(a):
 if a > 0:
 return '+' +str(a)
 elif a<0:
 return '-' +str(abs(a))
 elif a==0:
 return str(a)

def h(a, b, c):
 d = b**2 - 4*a*c
 if a == 0:
 print('Equation ' + str(b) +'x' + g(c) +' = 0' + ' is not square')

 else:
 if d > 0:
 x1 = (-b + pow(d, 1/2))/2/a
 x2 = (-b - pow(d, 1/2))/2/a
 print('Real roots are x1=' + g(x1) + ' x2=' + g(x2))
 elif d == 0:
 x1 = x2 = -b/2/a
 print('Real roots are x1= x2= ' + g(x2))
 elif d < 0:
 x11 = -b/2/a
 x12 = pow(abs(d), 1/2)/2/a
 x22 = -x12
 print('Complex roots are x1= ' + g(x11) + g(12) +'j'+ ' x2= ' + g(x11) + g(x22) +'j' + ' j**2 = -1')

 return

```

Результати розв'язування рівнянь при різних значеннях дискримінанту **d**:

```

>>> import first
>>> first.h(0, 4, 1)
Equation 4x +1 = 0 is not square
>>> first.h(1, 4, 4)
Real roots are x1= x2= -2.0
>>> first.h(1, 5, 4)
Real roots are x1=-1.0 x2=-4.0
>>> first.h(1, -4, 5)
Complex roots are x1= +2.0+12j x2= +2.0-1.0j j**2 = -1

```

Зауважимо, що символ **j** у Python коду є уявну одиницю.

### Інструкція **return** і значення, що повертаються функцією

Значення, що повертається функцією – це, об'єкт, який створюється функцією і повертається при виконанні інструкції з таким форматом:

**return <список виводу>**

Інструкція на виконання функції має вигляд:

**<var> = name (<список фактичних значень аргументів>),**

де **var** – ім'я змінної, що вказує на об'єкт, повернений функцією. Приклади наведені вище.

Можливий варіант, коли функція використовується без явного визначення такої змінної. **name (<список фактичних значень аргументів>)**

У цьому випадку об'єкт іменується анонімною змінною.

### **Приклад**

```
>>> def quad(a, b, c):
 d = b**2 - 4*a*c
 if a == 0:
 print('it''s no quad')
 else:
 return (-b + pow(d, 0.5)) / (2*a) , (-b - pow(d, 0.5)) / (2*a)

>>> quad(1, -4, -5)
(5.0, -1.0)
>>>
(5.0, -1.0)
```

Цей приклад також показує, що за ключовим словом **return** може слідувати список виразів, значення яких повертаються у програму.

Такий спосіб виклику функції може бути використаний і як аргумент іншої функції:

```
>>> s = pow(quad(1, -4, -5)[0], 0.5)
>>> s
2.23606797749979
>>>
```

У цьому прикладі обчислюється корінь квадратний із першого елементу списку розв'язків квадратного рівняння.

*А що поверне функція, якщо список після слова **return** буде пустим? 🤔*

**Іменовані аргументи і функція print ()** Більшість аргументів ідентифікуються за їх позиціями у викликах функції. Наприклад, виклик **random.randint(1, 10)** відрізняється від виклика **random.randint(10, 1)**. При виклику функції **random.randint(1, 10)** повертається випадкове ціле число з діапазону від 1 до 10, оскільки перший аргумент має зміст нижньої межі діапазону, а другий – верхньої. Виклик **random.randint(10, 1)** веде до помилки).

У той же час Python надає можливість іменувати параметри функції. Іменовані параметри є необов'язковими у тому розумінні, що їх можна опускати і функція використовує їх значення за замовчанням.

Наприклад, функція **print()** має необов'язкові параметри **end** та **sep**, за допомогою яких можна задати відповідно текст, що виводитиметься наприкінці аргументів, і текст, який повинен виводитися між аргументами (подільник).

```
>>> print()
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Якщо запустити програму (**Python** дозволяє писати інструкції рядок)

```
>>> print('Hello'); print('World')
Hello
World
```

Два рядки з'являться у вигляді окремих рядків виводу, оскільки функція **print( )** автоматично додає символ нового рядка (переведення каретки) наприкінці кожного рядка, який їй передається.

У випадку необхідності замість символу нового рядка (переведення) можна використати інший рядок, задав його за допомогою іменованого аргументу **end**. Наприклад, якщо використати порожній рядок, маємо

```
>>> print('Hello', end=''); print('World')
HelloWorld
```

Тут текст виводиться в одному рядку, оскільки відсутній символ нового рядка після тексту 'Hello'. Замість нього виводиться порожній рядок.

Цей прийом пригодиться вам у тих випадках, коли потрібна відміна використання символів порожніх рядків, які автоматично додаються у кінці кожного виводу за допомогою функції **print( )**.

Так само, якщо функції **print( )** передається декілька строкових значень, то по замовчанню у якості подільника використовується пробіл. Прикладом є такий код.

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

Проте, змістъ заданого за замовчанням порожнього рядка можна використати інше, передав функції іменований аргумент **sep**. Введіть код:

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

Іменовані аргументи також можна додавати при створені функції користувачем. Проте спочатку необхідно вивчити такі типи даних, як списки й словники, про які піде мова у двох наступних главах. Поки що достатньо знати лиши те, що у деяких функцій є іменовані аргументи, які можна задавати при виклику функції.

## **Локальна й глобальна області видимості**

Про параметри й змінні, які отримують значення у тілі викликаної функції говорять, що вони існують у локальній області видимості цієї функції.

Про змінні, значення яким присвоюється зовні функції, говорять, що вони існують у глобальній області видимості.

Змінні у локальній області видимості, називаються локальними змінними, тоді як змінні, які існують у глобальній області видимості, називаються глобальними змінними.

Змінна може відноситься **тільки** до одного з цих типів; вона не може бути локальною і глобальною одночасно.

Можна уявляти собі область видимості як контейнер для змінних. При знищенні області видимості всі значення, які зберігаються у змінних відповідного типу, також губляться.

Існує тільки одна глобальна область видимості, і вона створюється у момент початку виконання програми. Коли програма завершує роботу, глобальна область видимості знищується, і всі її змінні губляться. Інакше, при запуску програми у черговий раз змінні містили б ті ж самі значення, що і під час останнього сеансу виконання програми.

Локальна область видимості створюється кожного разу, коли викликається функція. Будь-яка змінна, якій надається значення в цій функції, існує у даній локальній області видимості. При поверненні із функції локальна область видимості знищується, і ці змінні губляться. Коли наступного разу викликається та ж сама функція, локальні змінні не будуть пам'ятати ті значення, які зберігалися в них при останньому виклику функції.

Області видимості змінних грають важливу роль за наступними причинами:

1. Локальні змінні не можуть використовуватися у коді, який відноситься до глобальної області видимості. У той же час локальна область видимості має доступ до глобальних змінних.
2. Код, який знаходиться у локальній області видимості функції, не може використовувати змінні із будь-якої іншої локальної області видимості.
3. Різні змінні можуть мати одне і то ж ім'я, якщо вони відносяться до різних областей видимості. Це означає, що одночасно можуть існувати як локальна змінна з певним іменем, так глобальна змінна з таким самим іменем.

Використання у **Python** різних областей видимості і відмова від того, щоб вважати всі змінні глобальними, забезпечує таку перевагу, що функції, які змінюють змінні, можуть взаємодіяти з рештою коду програми тільки через свої параметри й значення, що повертається.

При такому підході контролювати поведінку програми набагато легше й безпечніше.

Якщо б всі змінні у програмі були тільки глобальними, то помилкове значення будь-якої змінної могло б передаватися в інші частини програми, тем самим ускладнюючи пошук причини помилки. Значення глобальної змінної може встановлюватися у будь-якому місті програми, а вся програма може нараховувати тисячі рядків!

Якщо помилка пов'язана з невірним значенням локальної змінної, то для знаходження причини помилки достатньо дослідити лише код функції, у якій встановлюється значення даної змінної.

У невеликих програмах використанні глобальних змінних припустимо, але дотримуватися такого ж стилю у випадку великих програм – погана звичка.

Локальні змінні не можуть використовуватися у глобальній області видимості. Сутність такої архітектури Python ілюструє приклад

```
>>> def spam():
 global var
 eggs = 'Hi'
 var = 'Hi'

>>> spam()
>>> print(eggs)
Traceback (most recent call last):
 File "<pyshell#21>", line 1, in <module>
 print(eggs)
NameError: name 'eggs' is not defined
>>> print(var)
Hi
```

С початку визначається функція `spam()` без аргументів. У тілі функції використовуються дві змінні `eggs` та `var`.

Звертання до функції далі відбувається без проблем.

Проте спроба надрукувати значення `eggs` є помилкою, оскільки інтерпретатор працює з глобальною областю пам'яті програми, і не «бачить» змінну `eggs` у локальній області пам'яті функції.

У той же час, змінна `var` об'явлено у тілі функції, як глобальна, інтерпретатор її «бачить» і значення друкується. Це й ілюструє наступний фрагмент

```
>>> id(eggs)
Traceback (most recent call last):
 File "<pyshell#28>", line 1, in <module>
 id(eggs)
NameError: name 'eggs' is not defined
>>> id(var)
2615776282160
```

З цього випливає, що

**У локальних областях видимості не можуть використовуватися змінні із інших локальних областей видимості.**

Отже, природно, що в одній локальній області і деякій іншій локальній області можуть існувати змінні з однаковими іменами, проте це різні змінні. Це ілюструє наступний фрагмент

```
>>> def spam():
 global var
 eggs = 'Hi'
 var = eggs

>>> def nospam():
 global par
 eggs = 'Bey'
 par = eggs

>>> spam()
>>> nospam()
>>> print(var)
Hi
>>> print(par)
Bey .
```

## Контрольні питання

1. Що дає використання функцій у програмах?
2. Коли саме виконується код функції: коли вона визначається чи коли викликається?
3. За допомогою якої інструкції створюються функції?
4. Чим відрізняється визначення функції від виклика?
5. Скільки глобальних областей видимості може мати програма на мові Python?  
Скільки локальних?
6. Що відбувається із змінними, які знаходяться у локальній області видимості, при поверненні управління від функції до програми?
7. Що таке значення, що повертається? Чи може таке значення бути частиною виразу?
8. Яке значення повертається, якщо функції не містить інструкції return?
9. Що таке тип даних None?
10. Що робить інструкція import?
11. Якщо функція Васоп ( ) міститься у модулі spam, то як її викликати після імпортuvання цього модуля?

## Завдання:

1. Напишіть функцію **is\_odd**, яка отримує ціле число і повертає **True** для непарних чисел, або **False** для непарних.
2. Напишіть функцію **is\_prime**, яка отримує ціле число і повертає **True** для простих чисел або **False** для чисел, які не є простими (**Просте число таке, яке не має дільників, крім одиниці та себе**).
3. Напишіть функцію бінарного пошуку. Функція повинна отримувати відсортовану послідовність і шуканий елемент, а повернати індекс шуканого елементу. Якщо елемент не знайдено, функція повинна повернати **-1**.
4. Напишіть функцію, яка отримує рядки у «верблужому регістрі» (**ThisIsCamelCased**) і перетворювати їх у «зміїний регістр» (**this\_is\_camel\_cased**). Змініть функцію, додавши до неї аргумент **separator**, щоб функція також могла виконувати перетворення до «кебаб-регістру» (**this-iscamel-cased**).
5. Послідовність Коллатца

Напишіть функцію **collatz ( )**, яка приймає один параметр: **number**. Якщо **number** – парне число, функція **collatz ( )** повинна вивести на екран й повернути значення **number // 2**. Якщо **number** – непарне число, то функція повинна вивести на екран й повернути значення **3 \* number + 1**.

Після цього напишіть програму, яка пропонує користувачу ввести ціле число не менше двох, а потім послідовно викликає функцію **colltz ( )** для цього числа і значень, які повертаються черговим викликом цієї функції, доки на певному етапі не буде повернене значення **1**.

**Таким чином** буде проілюстрований простий факт, який досі не має строго математичного обґрунтування:

***Незалежно від вибору початкового числа програма все одно рано, чи пізно отримає одиницю!***

Числова послідовність, яку генерує програма називається послідовністю Коллатца, є однієї найпростіших за формою невирішених проблем математики.  
[https://en.wikipedia.org/wiki/Collatz\\_conjecture](https://en.wikipedia.org/wiki/Collatz_conjecture)

## Лабораторна робота 9 Тема:

### «Списки»

#### Мета:

1. Списки у **Python**, як упорядкована множина даних будь-якого типу.
2. Відповісти на контрольні питання та виконати завдання.
3. Виконати першу частину курсової роботи.

### Теоретичний мінімум

Ще одна тема, з якою слід познайомитися перед тим, як писати власні серйозні програми, – це списки й близький до нього тип даних - кортежи.

Списки й кортежи можуть містити упорядковану множину значень, що спрощує створення програм із оброблення великих об'ємів даних.

І оскільки списки самі можуть містити інші списки, надається можливість відтворення на програмному рівні одну з визначальних властивостей інформації – її ієрархічну структуру.

Тут наведені основні відомості про списки. Також мова піде про методи, пов'язані з цим типом даних. Далі розглядаються подібні типи даних, як кортежи та рядки, їх зв'язок із списками та чим вони принципово відрізняються.

Крім самостійного інтересу, цей матеріал є підготовчим для наступного, де мова піде про словники.

### Що таке список

Список – це структура даних, яка є упорядкованою послідовністю (колекцією) даних будь-якого типу, що визначені у **Python**. Доступ до структурних частин здійснюється за допомогою індексів

У прикладі список утворений рядком, цілім й десятковим числом, іншим списком й інструкцією, яка викликає на виконання описану заздалегідь функцію.

```
>>> def T(a, b=2):
 return a*b

>>> s = ['assa', 23, 3.14, [1,2], T(6), T(6, b=-2)]
>>> for i in range(6):
 print('s[',i,']=', s[i], sep=' ', end=' ')

s[0]=assa s[1]=23 s[2]=3.14 s[3]=[1, 2] s[4]=12 s[5]=-12
```

На цьому ж прикладі показано, як можна звертатися до структурних частин першого та вищих ієрархічних рівнів списку. Отже, список – індексна змінна:

```
>>> s[0:4]
['assa', 23, 3.14, [1, 2]]
>>> s[1:3]
[23, 3.14]
>>> s[3][0]
1
```

## Від'ємні індекси

Не дивлячись на те, що відлік індексів починається з нуля, у якості індексів дозволяється використовувати від'ємні значення.

Від'ємному значенню -1 відповідає останній елемент списку, значенню -2 – передостанній і т. п. Доцільність такої можливості продемонструємо простим прикладом.

У кореневому каталозі Python створений файл list.py, який містить список. Перший елемент списку має індекс 0, проте, треба звернутися до останнього елементу списку, а кількість елементів невідома:

```
>>> import list
>>> list.sp[-1]
'd'
```

## Доступ до частини списку за допомогою зрізу списку

Вище було показано, як за допомогою індексів отримати доступ до окремих елементів, а також частин списку. Доступ за допомогою індексів до частини списку називають «зрізом списку». У загальному випадку інструкція на отримання зрізу має вигляд

**list\_name[subscript : superscript]**

Результатом буде новий список, складений із елементів списку **list\_name** з індексами від **subscript** до **superscript-1** (див. попередні приклади). Припускається й скорочений запис зрізу, а саме:

- Якщо відсутній перший індекс, мається на увазі, що зріз починається із початку списку, тобто елементу з індексом 0.
- Якщо відсутній другий індекс, то мається на увазі, що останній елемент зрізу є останнім елементом списку.
- Природно, що відсутність і першого і другого індексів у зрізі (тільки дві крапки, ознака зрізу) означає, що зрізається увесь список. Всі ці випадки показані на прикладі

```

>>> def T(a, b=2):
 return a*b

>>> s = ['assa', 23, 3/14, [1,2], T(6), T(6, b=-2)]
>>> s[:4]
['assa', 23, 0.21428571428571427, [1, 2]]
>>> s[2:]
[0.21428571428571427, [1, 2], 12, -12]
>>> s[:]
['assa', 23, 0.21428571428571427, [1, 2], 12, -12]

```

## Зміна значень у списках за допомогою індексів

Індексація використовується і для зміни значення елементу списку:

```

>>> def T(a, b=2):
 return a*b

>>> def P(a, b = 3, c = 6):
 return a//b//c

>>> s = ['assa', 23, 3.14, [1,2], T(6), T(6, -2)]
>>> s
['assa', 23, 3.14, [1, 2], 12, -12]
>>> s[5]=P(123)
>>> s
['assa', 23, 3.14, [1, 2], 12, 6]

```

## Конкатенація і реплікація списків

Про ці операції мова йшла вище, проте повторимо. За допомогою операції + можна об'єднати два списку у новий список Списки можна й множити на ціле число, що означає відповідне повторення вмісту списку

```

>>> q=['a', 'b']
>>> w=['c', 'd', 'e']
>>> 3*(q + w)
['a', 'b', 'c', 'd', 'e', 'a', 'b', 'c', 'd', 'e', 'a', 'b', 'c', 'd', 'e']
>>> 3*q + 3*w
['a', 'b', 'a', 'b', 'a', 'b', 'c', 'd', 'e', 'c', 'd', 'e', 'c', 'd', 'e']

```

На прикладі показано, що операції конкатенації й реплікації списків не підпорядковані дистрибутивному закону алгебри.

## Видалення елементів із списку

Інструкція **del** видаляє із списку елемент із заданим індексом. Всі елементи, які знаходяться справа від цього елементу, зсуванося вліво на один індекс. Аналогічно, ця операція може видалити і зріз списку

```
>>> del(s[2])
>>> s
['assa', 23, [1, 2], 12, 6]
>>> del(s[1:3])
>>> s
['assa', 12, 6]
>>>
```

Інструкція **del** також може видалити із програми будь який об'єкт, як простий, так і структурований. У прикладі видаляється список.

```
>>> del(s)
>>> s
Traceback (most recent call last):
 File "<pyshell#38>", line 1, in <module>
 s
NameError: name 's' is not defined
```

## Ініціалізація списку з наперед невідомою кількістю елементів

Вище було показано, що список можна задати явно, переліком елементів.

Проте, часто зустрічається випадок, коли кількість елементів списку наперед не відома.

Наприклад, список формує користувач інтерактивним введенням елементів. У цьому і подібних випадках, може бути використана змінна, яка посилатиметься на порожній список `[ ]`. А потім, за допомогою методів **append** або **insert** (див. далі) цей список може бути доповнений необхідною кількістю елементів.

## Інструкції **in** та **not in**

Однією з поширених базових задач оброблення даних є визначення належності заданого об'єкту заданому списку. Для цього у Python використовуються інструкції **in** й **not in**. Оскільки ці інструкції, по суті є реалізацією бінарного відношення теорії множин, їх формат наступний

```
<element> in <list>
<element> not in <list>
```

Результатом обчислення цих виразів є булеві значення **True** або **False**.

```

>>> s = ['Kate', 'Sophy', 'Honey']
>>> 'Sly' in s
False
>>> 'Sly' not in s
True
>>> s.append('Sly')
>>> 'Sly' in s
True
>>> s
['Kate', 'Sophy', 'Honey', 'Sly']

```

## Групові операції

У Python реалізовані можливості, які підвищують продуктивність праці програміста за рахунок усунення необхідності виконання рутинних операцій. Однією з таких можливостей є так зване **групове присвоювання**:

```

>>> d, e, f, g = s
>>> d
'Kate'
>>> e, f
('Sophy', 'Honey')
>>> a, b, c = 12, 'abba', [1,2]
>>> c
[1, 2]

```

У прикладі зліва від операції присвоювання стоїть перелік імен змінних, яким надається значення. Справа – список цих значень. Вивід змінних демонструє значення, які воно набули.

У другому прикладі зліва також перелік змінних. Проте, справа також перелік об'єктів, причому різного типу. **Основне правило** групового присвоювання – кількість змінних зліва повинна **бути рівною** об'єктів справа.

Другою можливістю є **комбіноване присвоювання** (табл), що є узагальненням інкременту та декременту у, наприклад, мові C++. Правила таких операцій наведені у таблиці

| Присвоювання     | Комбіноване присвоювання |
|------------------|--------------------------|
| $var = var + a$  | $var += a$               |
| $var = var - a$  | $var -= a$               |
| $var = var * a$  | $var *= a$               |
| $var = var / a$  | $var /= a$               |
| $var = var // a$ | $var //= a$              |
| $var = var \% a$ | $var \%= a$              |

|  |  |
|--|--|
|  |  |
|--|--|

У таблиці **a** – об'єкт, тип якого відповідає об'єкту **var**. Отже операції виконуються відповідно до контексту виразу. На приклад, якщо **var** і **a** числа, то + операція додавання, а \* множення. Якщо **var** список, **a** – ціле додатне число, то \* операція реплікації і т. і.

## **Методи**

З точки зору структурного програмування, програма є сукупністю функцій або процедур, які за певним алгоритмом переробляють вхідні дані у кінцеві цільові дані.

З точки зору об'єктно-орієнтованого програмування, програма є сукупністю об'єктів, які за певним алгоритмом змінюють свій початковий стан у кінцевий цільовий стан.

Стан об'єкту змінюють методи. По суті, метод є функцією (**але не навпаки!**).

Різниця полягає у наступному. Об'єкт є значенням, екземпляром класу. Клас – це інкапсуляція типу даних і функцій, які є застосованими тільки до екземплярів класу. Такі функції і називають **методами**.

Отже, по-перше, якщо клас описаний у програмі, то нема необхідності описувати методи цього класу. І по-друге, враховуючи цю обставину, синтаксис інструкції із виклику методу відрізняється від виклика функції.

І по третє, основне, результатом оброблення даних функції є нові дані. Метод **не створює** нового об'єкту. Він може змінити стан, або надати інформацію про стан об'єкту.

Докладніше цей матеріал розглядається далі. Нижче наводяться декілька поширеніших методів для списків.

## **Пошук значення у списку методом **index()****

Списки мають метод **index()**, який приймає значення, проглядає список і повертає індекс першого подібного елементу, що зустрінеться. Якщо вказане значення відсутнє у списку, то інтерпретатор **Python** генерує помилку **ValueError**.

```
>>> s = [23, 3.14, 'assa', [1,2], 'assa']
>>> s.index('assa')
2
>>> s.index('abba')
Traceback (most recent call last):
 File "<pyshell#4>", line 1, in <module>
 s.index('abba')
ValueError: 'abba' is not in list
```

## Методи append й insert ( ) додавання значень у список

Якщо необхідно додати нові елементи у список, використовуються методи `append()` й `insert( )`. Перший записує у кінець списку, а другий записує після елемента списку, індекс якого є параметром методу.

```
>>> s = [23, 3.14, 'assa', [1,2], 'assa']
>>> s.append('baab')
>>> s
[23, 3.14, 'assa', [1, 2], 'assa', 'baab']
>>> s.insert(2,'baab')
>>> s
[23, 3.14, 'baab', 'assa', [1, 2], 'assa', 'baab']
>>> s.insert(-1, 'baab')
>>> s
[23, 3.14, 'baab', 'assa', [1, 2], 'assa', 'baab', 'baab']
>>> spam = s.insert(-1, 'baab')
>>> print(spam)
None
```

Цей приклад показує, що методи `append()` й `insert(-1, )` діють однаково. Останні два рядки прикладу показують, методи дійсно не створюють нового значення, а тільки змінюють стан списку.

Отже, методи інкапсульовані у клас і застосовані тільки для екземплярів цього класу

```
>>> egges = 'Whisky'
>>> egges.insert(0, 'Irish')
Traceback (most recent call last):
 File "<pyshell#23>", line 1, in <module>
 egges.insert(0, 'Irish')
AttributeError: 'str' object has no attribute 'insert'
```

## Видалення значень із списку методами remove( ) й інструкції del

Методу `remove( )` передається об'єкт, який треба видалити із списку, для якого він викликається. При спробі видалити об'єкт, який відсутній у списку, виникає помилка `ValueError`. Якщо у списку декілька одинакових елементів з різними індексами, то буде видалений перший з них, що зустрінеться при перегляду списку зліва на право.

Якщо відомий індекс елемента, то для видалення можна використовувати інструкцію **del**. Всі ці дії демонструє приклад:

```
>>> s = [23, 3.14, 'assa', [1, 2], 'assa']
>>> s.remove('assa')
>>> s
[23, 3.14, [1, 2], 'assa']
>>> del s[1]
>>> s
[23, [1, 2], 'assa']
>>> s.remove(3.14)
Traceback (most recent call last):
 File "<pyshell#19>", line 1, in <module>
 s.remove(3.14)
ValueError: list.remove(x): x not in list
```

### Сортування значень у списки методом sort()

Для сортування елементів у списку використовується метод **sort()**. Природно, що лексикографічне упорядкування можливе на даних одного типу. Отже, метод сортує **тільки списки, які утворені елементами одного типу даних**. При спробі сортуванні списку, який містить дані різних типів, виникає помилка **TypeError**.

```
>>> s = [23, 3.13, [1,2]]
>>> s.sort()
Traceback (most recent call last):
 File "<pyshell#1>", line 1, in <module>
 s.sort()
TypeError: '<' not supported between instances of 'list' and 'float'
>>> s = [23, 3.13, 1.2]
>>> s.sort()
>>> s
[1.2, 3.13, 23]
>>> t = ['abba', 'tree', 'assa', 'y/n', 'you']
>>> t.sort()
>>> t
['abba', 'assa', 'tree', 'y/n', 'you']
```

Щоб сортувати список в оберненому порядку, треба вказати іменований аргумент **reverse** зі значенням **True**

```
>>> t.sort(reverse=True)
>>> t
['you', 'y/n', 'tree', 'assa', 'abba']
```

При сортуванні рядків треба враховувати наступне. Оскільки таблиця кодів **ASCII** з'явилася раніше, ніж таблиця **ASCII**, а у першій усі символи вважалися як заголовні літери, і друга є, фактично, продовженням першої, то метод **sort()** сортує рядки не у алфавітному порядку, а у відповідності до таблиці **ASCII**. Це означає, що букви заголовні (верхній регістр) передують буквам у нижньому

регистрі. Тому, наприклад, буква **a** буде розташована у процесі сортування після букви **Z**

```
>>> t.append('Zuma')
>>> t.append('Alice')
>>> t
['you', 'y/n', 'tree', 'assa', 'Zuma', 'Alice']
>>> t.sort()
>>> t
['Alice', 'Zuma', 'assa', 'tree', 'y/n', 'you']
```

Це не дуже зручно. Щоб, хоча б у певній мірі, компенсувати такі незручності, метод **sort** має необов'язковий аргумент **key**. За замовчанням він відсутній і метод виконує сортування із вказаними особливостями. Якщо цей параметр задати явно із значенням **str.lower**, то усі рядки будуть сортуватися так, ніби вони записані літерами у нижньому регистрі, не змінюючи значення самих літер. Якщо продовжити попередній приклад, отримуємо

```
>>> t.sort(key=str.lower)
>>> t
['Alice', 'assa', 'tree', 'y/n', 'you', 'Zuma']
```

## Завдання

### Навчальний проект (курсова робота, частина 1)

Використовуючи розглянутий матеріал, можна написати програму, яка буде сортувати списки з елементами різного типу даних. Сортування здійснюється на підставі відношення порядку і тут можливі три підходи:

#### *Більш просунутий:*

Придумати і визначити відношення порядку на множині всіх основних типів даних Python, тобто ‘int’, ‘float’, ‘str’, ‘list’, і написати відповідну програму.

#### *Менш просунутий:*

Програма перевіряє тип даних елементів списку і сортує між собою елементи одного типу.

#### *Природний:*

Програма сортує у списку дані заданого типу. Якщо даних такого типу у списку немає, програма повертає повідомлення “**TypeError**” .

## Лабораторна робота 10

### Тема: «Списки, рядки і кортежи»

*Мета:*

1. Засвоїти фундаментальне для **Python** поняття базових (або неділімих) типів даних і структур даних. Розуміти різницю їх архітектури і методів оброблення.
2. Набути навички роботи з ними.
3. Відповісти на контрольні питання та виконати завдання.

### Теоретичний мінімум

У мовах високого рівня визначені **базові типи даних** (цілі та дійсні числа, символи, рядки і т. і.), які відображають певні базові, тобто неділімі, якості реальних об'єктів, і **структурі даних** (масиви, списки, множини і т. і.), які є певним чином організованими сукупностями даних базових типів і відображають властивості інформації (часткова, або лінійна упорядкованість, ієрархія даних і т. і.).

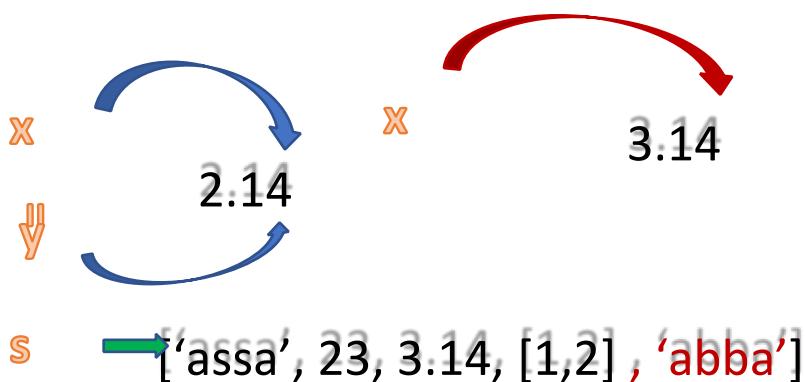
В результаті оброблення об'єкту базового типу утворюються новий об'єкт. Це можна проілюструвати простим прикладом:

```
>>> x = 2.55
>>> id(x)
2504292770416
>>> x = x + 0
>>> id(x)
2504293177904
```

Утворюється число 2.55 – об'єкт базового типу **float**. Змінна **x** зберігатиме посилання на цей об'єкт.

Виконується операція додавання, причому додається 0 і значення числа не змінюється. Проте результат – новий об'єкт, оскільки змінна **x** зберігатиме посилання на інший об'єкт. Куди поділося число 2.55?

При обробленні структур даних, як правило, операції виконуються з деякими частинами такого об'єкту. Решта не змінюється. Тобто, змінюється стан об'єкту. Природно, що при цьому ідентифікатор, адрес об'єкта, не змінюється. Нижче наведено рисунок, який ілюструє сказане і код



```

>>> x = 2.14
>>> y=x
>>> y
2.14
>>> id(x)
1438680162928
>>> id(y)
1438680162928
>>> x = x+1
>>> x
3.14
>>> id(x)
1438680166288
>>> y
2.14
>>> id(y)
1438680162928
>>> s = ['assa', 23, 3.14, [1,2]]
>>> id(s)
1438681339328
>>> s.append('abba')
>>> s
['assa', 23, 3.14, [1, 2], 'abba']
>>> id(s)
1438681339328

```

Після присвоювання, змінні **x** та **y** зберігають посилання на один і той же об'єкт типу **float**. Після додавання утворюється новий об'єкт, на який посилається **x**.

Список є структурою даних і додавання нового елементу – це просто зміна його стану. Посилання не змінюється.

Нагадаємо, базові типи даних є неділімими і після їх обробленні утворюються нові об'єкти. **Python**-програміст, природно, повинен про це знати і враховувати у своїй роботі.

При розв'язуванні багатьох задач зручним поданням даних є рядки і кортежи, які з одного боку, дуже схожі на списки, а з іншого, є базовими, неділімими, типами даних і обробляються відповідно.

## Рядки

Списки – не єдиний тип даних, який являє собою упорядковані послідовності значень. Наприклад, рядки є упорядкованою послідовністю символів.

У наслідок цього, багато з того, що можна робити зі списками, можна робити і з рядками: індексування, зрізи, а також використання у циклах `for` як множину значень переліченої змінної і т.і. На приклад:

```
>>> r = 'AbbA'
>>> index = 0
>>> for var in r:
 print(' Рядок містить символ ' + r[index])
 index += 1
```

```
Рядок містить символ А
Рядок містить символ б
Рядок містить символ б
Рядок містить символ А
```

Проте, відміна полягає у тому, що рядок зберігається як цілісний об'єкт. І спроба змінити його, як структурований об'єкт, викликає помилку:

```
>>> r
'AbbA'
>>> id(r)
2537298225008
>>> r[0]
'A'
>>> r[0] = 'C'
Traceback (most recent call last):
 File "<pyshell#13>", line 1, in <module>
 r[0] = 'C'
TypeError: 'str' object does not support item assignment
>>> r += 'd'
>>> r
'AbbAd'
>>> id(r)
2537298224816
```

Цей же приклад показує, якщо виконати операцію із рядком, наприклад, конкатенацію, то змінюється не стан об'єкту, а утворюється новий об'єкт. Попередній губиться, оскільки та ж змінна тепер містить інше посилання.

Нижче показано, що подібні операції із структурованим об'єктом змінюють його стан. Новий об'єкт не створюється.

```
>>> s = ['A', 'b', 'b', 'A']
>>> id(s)
2537298532352
>>> s += 'D'
>>> s
['A', 'b', 'b', 'A', 'D']
>>> id(s)
2537298532352
```

## Кортежи

Кортеж це вираз мови **Python** у вигляді послідовності інших виразів мови, правильних синтаксично у контексті логіки програми і яка обмежена круглими дужками.

Отже, зовні, з точки зору синтаксису, кортежи майже ідентичні спискам і відрізняються тільки межами, круглими, а не квадратними дужками:

```
>>> a = [1, 'g', f(1), (1, 'sos'), [1, 3]]
>>> b = (1, 'g', f(1), (1, 'sos'), [1, 3])
>>> type(a)
<class 'list'>
>>> type(b)
<class 'tuple'>
```

Відповідно, кортежи, як і списки, є індексованими послідовностями. Їх можна використовувати як перелічену змінну у циклах, застосовувати операції із індексами, наприклад зрізи, і таке інше:

```
>>> y = 5
>>> for i in b:
 y +=2
 print(y)
```

```
7
9
11
13
15
```

```
>>> a[:2]
[1, 'g']
>>> b[:2]
(1, 'g')
```

Але, головною відміною кортежів від списків є те, що кортежі, подібно рядкам, відносяться до незмінних типів даних. За допомогою індексів можна отримувати доступ до структурних частин кортежу. Проте спроба змінити структурну частину кортежу веде до критичної помилки і зупинки програми:

```
>>> b[1]
'g'
>>> b[1] = 'h'
Traceback (most recent call last):
 File "<pyshell#22>", line 1, in <module>
 b[1] = 'h'
TypeError: 'tuple' object does not support item assignment
```

У той же час змінити список цілком можливо:

```
>>> a = [1, 'g', f(1), (1, 'sole'), [1, 2]]
>>> id(a)
2185370264192
>>> a[1] = 'h'
>>> a
[1, 'h', True, (1, 'sole'), [1, 2]]
>>> id(a)
2185370264192
```

Можна бачити, що адреса списку не змінилася. Тобто, це той самий об'єкт у іншому стані.

Спробуємо змінити кортеж **b** з нашого прикладу. Визначимо його адресу у RAM. Додаєм до кортежу елемент. Отже, кортеж змінився, а його адреса ні? Але якщо вивести на екран значення кортежу **b**, то побачимо, що він не змінився, як і адреса. А новий кортеж став значенням анонімної змінної, якщо не забули, що це таке.

Використаємо операцію присвоювання. Дійсно, значення змінилося, але **b** – це зовсім інший об'єкт, оскільки змінилася його адреса у RAM. Попередній об'єкт **b** загублено.

```
>>> b = (1, 'g', f(1), (1, 'sole'), [1, 2])
>>> id(b)
2185369798592
>>> b + (5,)
(1, 'g', True, (1, 'sole'), [1, 2], 5)
>>> id(b)
2185369798592
>>> b
(1, 'g', True, (1, 'sole'), [1, 2])
>>> b = b + (5,)
>>> b
(1, 'g', True, (1, 'sole'), [1, 2], 5)
>>> id(b)
2185370282688
```

Лишилося прокоментувати вираз (5, ). За контекстом, ми хочемо «зчепити» два коржи, один з яких містить тільки один елемент.

Кома після елементу – це пояснення інтерпретатору, що має справу зі структурою даних, з кортежом з одного елементу, тобто з послідовністю з одного елементу, а не з цілим числом:

```
>>> type((5,))
<class 'tuple'>
>>> type((5))
<class 'int'>
```

## Примітка

Для чого потрібні незмінні типи даних, зокрема, кортежи і рядки?

1. *Захист від дурня (пробачте).* Кортежи і рядки, як правило, це опис даних, які не передбачається змінювати, а випадкова зміна веде до критичних наслідків. Наприклад, початкові дані.
2. *Займають суттєво менший об'єм RAM, ніж списки:*

```
>>> a = [1, 'g', f(), (1, 'sole'), [1, 2]]
>>> a.__sizeof__()
80
>>> b = (1, 'g', f(), (1, 'sole'), [1, 2])
>>> b.__sizeof__()
64
```

Та інші можливості, які стають очевидними при набутті досвіду програмування.

### **Коментар**

Подвійне підкреслювання – це позначення *власного методу класу*. Ця властивість Python надає можливість надавати методам однакові імена, якщо вони виконують подібні дії, але над екземплярами різних класів.

### **Перетворення типів функціями `list()` й `tuple()`**

Об'єкти типу **tuple**, кортежи, відрізняються від об'єктів типу **list**, списки, по суті, заборонено на зміну їх стану. Отже доцільно мати функцію, яка утворює спроможну до зміни копію об'єкту, водночас, зберігаючи недоторканим зразок даних. Такою функцією є `list()`. Природно, що обернене перетворювання списку у кортеж здійснює функція `tuple()`:

```
>>> a = ('Mary Twain', '24 years old')
>>> type(a)
<class 'tuple'>
>>> a[1] = '18 years old'
Traceback (most recent call last):
 File "<pyshell#11>", line 1, in <module>
 a[1] = '18 years old'
TypeError: 'tuple' object does not support item assignment
>>> b = list(a)
>>> type(b)
<class 'list'>
>>> b[1] = '18 years old'
>>> b
['Mary Twain', '18 years old']
>>> a = tuple(b)
>>> a
('Mary Twain', '18 years old')
```

### **Ще раз про посилання на типи даних, що змінюються і не змінюються**

У всіх мовах програмування високого рівня змінні іменують певні області RAM, у яких зберігаються об'єкти програми.

Вище було показана особливість архітектури мов програмування надвисокого рівня (VHLL) – змінні іменують не області RAM, у яких

зберігаються об'єкти програми, а комірки, де зберігаються посилання на ці об'єкти. Тобто, змінна, яка іменує, наприклад, список, містить не список, а посилання на список (адресу списку у RAM).

У прикладі рядок перетворюється на список. Новостворений список виводиться на екран двічі: з використанням змінної, що його іменує і за допомогою адреси списку, який зберігається у цій змінній. Для цього використовується модуль `ctypes`, який зберігає необхідні методи.

```
>>> import ctypes
>>> a_str = 'Hello, Python'
>>> a_list = list(a_str)
>>> print(a_list, end='\n')
['H', 'e', 'l', 'l', 'o', ',', ' ', 'P', 'y', 't', 'h', 'o', 'n']
>>> id(a_list)
1753365750592
>>> print(ctypes.cast(id(a_list), ctypes.py_object).value)
['H', 'e', 'l', 'l', 'o', ',', ' ', 'P', 'y', 't', 'h', 'o', 'n']
```

Нагадуємо, що адреса об'єкту може бути отримана за допомогою функції `id()` і для зручності користувача має вигляд звичайного цілого десяткового числа, хоча фізичні адреси об'єктів – це шістнадцятирічні числа. Фактичне значення адреси об'єкту із нашого прикладу може бути отримано так

```
>>> id(a_str) # decimal inteder
1753365424688
>>> hex(id(a_str)) # de facto
'0x1983cb92a30'
```

Нагадаємо (але незадовбуюмо!), якщо змінна посилає на об'єкт незмінного типу даних (float, str, tuple etc.), то присвоювання їй іншого значення веде до утворення нового об'єкту, що демонструє приклад

```
>>> s = t = 3.14
>>> s_id = id(s)
>>> t_id = id(s)
>>> s = t = 2.72
>>> print(id(s), ' ', id(t), end='\n')
2651992508080 2651992508080
>>> s = 2.72
>>> print(id(s), ' ', id(t), end='\n')
2651992906480 2651992508080
```

Змінна `t`, як і раніше, зберігає посилання на початковий об'єкт. Змінна `s`, тепер зберігає посилання на зовсім інший новостворений об'єкт.

При зміні стану об'єкту посилання на нього, природно, не змінюється:

```
>>> person = {'Name': ['Bob', 'Alice'], 'Age': [23, 18]}
>>> person = man = ['Bob', 'Giggs', 23]
>>> person[2] = 32
>>> print(person, ' ', man)
['Bob', 'Giggs', 32] ['Bob', 'Giggs', 32]
```

У прикладі змінні **person** і **man** містять посилання на один і той список.

### Копіювання об'єктів, функції **copy( )** і **deepcopy( )** модуля **copy**

При розроблені програм часто виникає необхідність зберігати копії таких об'єктів, як списки, словники і т. і., які б слугували певними зразками у подальшому. Це можна зробити функцією **copy** модуля **copy**.

```
>>> import copy
>>> u = [1, ['a', 3]] # template
>>> v = copy.copy(u)
>>> print(id(u), ' ', id(v))
2651993788992 2651993789376
>>> v[0] = 22
>>> print(u, ' ', v)
[1, ['a', 3]] [22, ['a', 3]]
```

З прикладу видно, що функція **copy** утворює новий список і при зміні копії **v** зразок **u** не змінився, т.к. змінні **u** і **v** посилають на різні об'єкти.

Проте, продовжимо, змінюючи стан списку, який є другим елементом списку **v**

```
>>> v[1][1]= ['abba', (1,2)]
>>> print(u, ' ', v)
[1, ['a', ['abba', (1, 2)]]] [22, ['a', ['abba', (1, 2)]]]
>>> print(id(u), ' ', id(v))
2651993788992 2651993485248
```

Можна побачити, що змінився не тільки «робочий» список **v**, але й зразок **u**. Чому? Спробуйте відповісти.

**Якщо не вдалося відповісти, то:**

Списки, словники, математичні вирази і т.і. мають структуру дерева. Функція **copy** утворює нові посилання тільки для вершин першого рівня. Для вершин нижніх рівнів зберігаються ті ж посилання, що й у зразку.

Отже, щоб зберігати недоторканим зразок ієрархічного об'єкту, копіювання треба виконувати функцією **deepcopy**. Виконаємо тіж дії з використанням функції **deepcopy( )** замість **copy( )** і зразок **u** не зміниться

```
>>> import copy
>>> u = [1, ['a', 3]] # template
>>> v = copy.deepcopy(u)
>>> print(id(u), ' ', id(v))
2503474728192 2503474727744
>>> v[0] = 22
>>> print(u, ' ', v)
[1, ['a', 3]] [22, ['a', 3]]
>>> v[1][1]= ['abba', (1,2)]
>>> print(u, ' ', v)
[1, ['a', 3]] [22, ['a', ['abba', (1, 2)]]]
```

Суть цього важливого матеріалу у наступному:

1. З точки зору дискретної математики, список є деревом, у вузлах якого знаходяться об'єкти будь-якого типу.
2. Така структура відповідає ієрархічній структурі інформації про реально існуючі об'єкти.
3. Перетворення списку веде тільки до зміну стану цієї структури. Її адреса у RAM при цьому не змінюється.
4. Адреса списку зберігається у змінній, яка іменує список. Складові списку також мають свої незалежні адреси у RAM. Ця обставина може бути джерелом помилок і її треба враховувати при копіюванні списків і подальшому використанні цих копій.
5. Ця ж обставина може є основою для принципово нової парадигми програмування, про створення і дослідження якої фахівці тільки наближаються.

### ***Примітка***

1. Зрозуміло, що наведених методів для конструктивної реалізації проектів. У таблиці всі методи для роботи зі списками ті їх призначення.

Це дасть змогу знайти на інтернет ресурсах необхідну інформацію.

### **Таблиця**

| Метод                   | Що робить                                          |
|-------------------------|----------------------------------------------------|
| <b>list.append(x)</b>   | Додає елемент <b>x</b> у кінець списку             |
| <b>list.extended(L)</b> | Додає у кінець списку всі елементи списку <b>L</b> |

|                                       |                                                                                                             |
|---------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <b>list.insert(i, x)</b>              | Вставляє у <b>i</b> -ту позицію елемент <b>x</b>                                                            |
| <b>list.remove(x)</b>                 | Видаляє першій, що зустрінеться елемент <b>x</b> . Якщо <b>x</b> немає у списку, повертає <b>ValueError</b> |
| <b>list.pop( [i] )</b>                | Видаляє <b>i</b> -ий елемент і повертає його. Якщо індекс не вказаний, видаляє останній елемент.            |
| <b>list.index(x, [start [, end]])</b> | Повертає положення першого елементу <b>x</b> (пошук ведеться від <b>start</b> до <b>end</b> )               |
| <b>list.count(x)</b>                  | Повертає кількість елементів <b>x</b> у списку                                                              |
| <b>list.sort([key=функція])</b>       | Сортує список за функцією                                                                                   |
| <b>list.reverse()</b>                 | Розвертає список                                                                                            |
| <b>list.copy()</b>                    | Копія списку (аналог <b>copy.copy()</b> )                                                                   |
| <b>list.clear()</b>                   | Очищає список                                                                                               |

2. Як створити список? Python не містить спеціальних методів. Проте, лаконічність та повнота цієї мови дає змогу організувати цей процес оптимально у кожній конкретній ситуації. Як приклади можна навести наступне:

- 2.1. Порожній список. Створити порожній список і заповнити його, використовуючи методи **append()**, **insert()** та **remove()**. Приклади наведені вище.
- 2.2. Перетворення типу. При необхідності, рядок можна перетворити на список. Приклади наведені вище.
- 2.3. Генератори списків. Python дає змогу використовувати цикли для створення, так званих, генераторів списків:

```
>>> r = '12345678'
>>> s = ['a' + i for i in r]
>>> print(s)
['a1', 'a2', 'a3', 'a4', 'a5', 'a6', 'a7', 'a8']
```

## Контрольні питання

1. Що означають дужки [ ] ?
2. Як присвоїти значення ‘abba’ у якості третього елементу списку [2, 4, 6, 8, 10]

У наступних трьох питаннях вважається

`s = ['a', 'b', 'c', 'd']`

3. Яке значення виразу `s[int('3' * 2) / 11]`? Обґрунтувати відповідь.

4. Яке значення виразу `s[-1]`?

5. Яке значення виразу `s[:2]`?

У наступних трьох питаннях вважається `s`

`= [3, 14, 'cat', 11, 'cat', True].`

6. Яке значення виразу `s.index('cat')`? 7. Як буде виглядати список після наступного виклику: `s.append(99)` 8. Як буде виглядати список `s` після наступного виклику:

`s.remove('cat')` 9. Які операції використовуються для конкатенації і реплікації списків?

10. У чому різниця між методами `append()` і `insert()`?

11. Назвіть два способи видалення значень із списків.

12. Назвіть декілька спільних ознак списків і рядків.

13. Чим кортежі відрізняються від списків?

14. Як записати кортеж, який мстить одне значення у вигляді цілого числа 42?

15. Як перетворити список у кортеж і навпаки.

16. Дано список

`s = [3, 14, 'cat', 11, 'cat', True]` Що містить комірка RAM з

ім'ям `s`. 17. Яке призначення функції `copy.copy()` і

`copy.deepcopy()`? Чим вони принципово відрізняються

## Завдання

### 1. Кома у якості подільника

Написати функцію, яка приймає список у якості аргументу і повертає рядок, у якому всі елементи списку подані символами, поділені між собою комами і пробілом, а перед останнім елементом вставлено слово `and`.

Наприклад:

`s = ['Father', 'mother', 'sister', 'brother', 'I']` à

'Father, mother, sister, brother and I'

Але функція повинна працювати з будь-якими списками.

### 2. Малювання символами

Припустимо, є список списків, у якому кожне значення внутрішніх списків являє собою рядок, складений із одного символу, як у наведеному прикладі

```
[['.', '.', '.', '.', '.', '.', '.'],
 ['.', '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', '0', '0', '0', '0', '.'],
 ['0', '0', '0', '0', '0', '0', '.'],
 ['.', '0', '0', '0', '0', '0', '.'],
 ['.', '.', '.', '.', '.', '.']]
```

У такому поданні список можна уявити як матрицю, а кожний елемент, як символ з координатами (номер стовпчика, номер рядка). Початок системи координат (0, 0) знаходиться у верхньому лівому куту, абсциса збільшується зліва на право, а ордината – зверху вниз.

- 1) Напишіть код, який подає фігуру на рисунку у вигляді списку.
- 2) Придумайте свою фігуру. Зробіть копію списку із попереднього завдання.

Змініть новоутворений список, щоб утворилася фігура, придумана вами. 3)

Надрукуйте зразок і вашу фігуру.

# Лабораторна робота № 11

## Тема: «Словники і структуровані дані»

### Мета:

1. Розібратися із словниками, структурою, яка широко використовується у машинному навчанні для подання даних.
2. Набути деякі навички роботи з ними.
3. Відповісти на контрольні питання, виконати завдання 4. Виконати другу частину курсової роботи.

### Теоретичний мінімум

У машинному навчанні «образ» реального об'єкту буде створюватися на основі так званої фреймової моделі.

Структура фрейму наступна:

| Фрейм (образ об'єкту) |                  |
|-----------------------|------------------|
| Слот 1                | Значення слоту 1 |
| Слот 2                | Значення слоту 2 |
| ....                  | ....             |
| Слот N                | Значення слоту N |

Слоти – це властивості або атрибути класу об'єктів. Значення слоту – це значення властивості або атрибуту даного, конкретного об'єкту цього класу. Природно, що значенням слоту можуть бути інші фрейми.

### Класичний приклад: Квітка ірису



| Iris setosa<br>label 0  |                        | Iris virginica<br>label 1 |                    | Iris versicolor<br>label 2 |
|-------------------------|------------------------|---------------------------|--------------------|----------------------------|
| Довжина<br>чашелистника | Ширина<br>чашелистника | Довжина<br>пелюстки       | Ширина<br>пелюстки | Вид<br>ірису               |
| 7.7                     | 3.8                    | 6.7                       | 2.2                | Iris virginica             |
| 4.9                     | 3.0                    | 1.4                       | 0.2                | Iris setosa                |
| 5.9                     | 3.0                    | 4.2                       | 1.5                | Iris versicolor            |
| 6.0                     | 2.2                    | 4.0                       | 1.0                | Iris versicolor            |
| 6.3                     | 3.3                    | 6.0                       | 2.5                | Iris virginica             |
| 5.2                     | 3.5                    | 1.5                       | 0.2                | Iris setosa                |
| 5.7                     | 2.9                    | 4.2                       | 1.3                | Iris versicolor            |

Образом квітки ірису є сукупність характеристик (рисунок), які у цілому подаються фреймом.

## Словники

Словники – тип даних, який забезпечує гнучкі можливості доступу до даних і ефективну організацію перетворень задач самої різної тематики.

Як і список, словник – це колекція значень об'єктів різних типів даних. Проте у словниках, на відзнаку від списків, індексами можуть бути не тільки цілі числа, але й дані іншого базового типу.

Індекси у словниках називаються ключами, а ключ разом із асоційованим з ним значенням – парою «ключ-значення».

Вираз, що задає словник у програмі має такий загальний вигляд

Name = {‘Key1’: value, ‘Key2’: value, ..., ‘KeyN’: value}

З точки зору машинного навчання, словники – ідеальна структура даних **Python** для подання образу обєкта на основі фреймової моделі.

При поданні фрейму словником індексами слугують імена слотів.

## **Приклад 1**

Класифікування квіток ірису за сортами є класичним прикладом машинного навчання. Квітка ірису подіється фреймом (див. вище). Навчальна вибірка – фрейм, у якому характеристикам ірису поставлені у відповідність відомі значення цільової функції – ім'я класу (сорт), до якого належить квітка. Навчальна вибірка зберігається, у вигляді словника, зберігається у бібліотеці **scikit-learn**. У програмі фрейм набуває ім'я **iris\_dataset**. За допомогою методу **.keys()** можна отримати перелік імен слотів цього фрейму

```
from sklearn.datasets import load_iris
iris_dataset = load_iris()
print('Keys iris_dataset: \n{}'.format(iris_dataset.keys()))
Keys iris dataset:
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'filename'])
```

## **Приклад 2**

У цьому прикладі наведені цілком реальні дані з пухлини молочної залозі, отримані у клініці Університету Вісконсин (США).

Ці дані також зберігаються у бібліотеці **scikit-learn** і їх дозволяється використовувати у навчальних цілях. Введемо ці дані

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
```

і отримаємо імені слотів

```
>>> print('Keys cancer: \n{}'.format(cancer.keys()), '\n')
Keys cancer:
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'filename'])
```

Імена слотів ( ключі ) і є індексами, за допомогою яких отримується доступ до значення слотів

```
>>> print(cancer['data'][:2], '\n')
[[1.799e+01 1.038e+01 1.228e+02 1.001e+03 1.184e-01 2.776e-01 3.001e-01
 1.471e-01 2.419e-01 7.871e-02 1.095e+00 9.053e-01 8.589e+00 1.534e+02
 6.399e-03 4.904e-02 5.373e-02 1.587e-02 3.003e-02 6.193e-03 2.538e+01
 1.733e+01 1.846e+02 2.019e+03 1.622e-01 6.656e-01 7.119e-01 2.654e-01
 4.601e-01 1.189e-01]
[2.057e+01 1.777e+01 1.329e+02 1.326e+03 8.474e-02 7.864e-02 8.690e-02
 7.017e-02 1.812e-01 5.667e-02 5.435e-01 7.339e-01 3.398e+00 7.408e+01
 5.225e-03 1.308e-02 1.860e-02 1.340e-02 1.389e-02 3.532e-03 2.499e+01
 2.341e+01 1.588e+02 1.956e+03 1.238e-01 1.866e-01 2.416e-01 1.860e-01
 2.750e-01 8.902e-02]]
```

або

```
>>> print(type(cancer['data']))
<class 'numpy.ndarray'>
>>> print(type(cancer['filename']))
<class 'str'>
```

## Завдання 1:

Визначити тип значень кожного слоту фрейму у попередніх прикладах. і отримати декілька значень даних.

## Завдання 2:

А)За власним бажанням визначитися із маркою авто і побудувати фрейм «Автомобіль» (дані взяти на офіційних сайтах дилерів в Україні). Подати фрейм словником. Б) Побудувати декілька запитів для отримання даних про авто за ключами.

## Завдання 3:

Побудувати словник, у якій містить дні народження ваших друзів, близьких та рідних. Навести приклади запитів до цієї БД.

## Порівняння словників і списків

Першим елементом у списку був би елемент з індексом **0**. У словниках поняття «перший елемент» немає. На відзнаку від списків, словники не упорядковані (**чому?**) структури даних. І, як наслідок, вони не припускають створення зразків, як це можливо у списках.

Отже, доступ до структурних частин словника можливий тільки за ключем. Якщо запит містить відсутній ключ, виникає помилка

```
>>> BD_my_friends = {'Bob': 'Jan 15', 'Tania': 'Jan 25',
... 'Samanta': 'April 2', 'Ivan': 'June10'}
>>> BD_my_friends ['Samanta']
'April 2'
>>> BD_my_friends ['Paul']
Traceback (most recent call last):
 File "<pyshell#12>", line 1, in <module>
 BD_my_friends ['Paul']
KeyError: 'Paul'
```

Проте, аналогічно виникає помилка IndexError при спробі виходу за межі припустимого діапазону індексів списку.

## **Методи для роботи із словниками**

### ***Методи keys () , values () и items ()***

При роботі із словниками часто використовуються методи **keys ()** , **values ()** й **items ()** . Ці методи повертають, відповідно, ключи, значення і пари “ ключ – значення” у вигляді даних спеціальних типів **dict\_keys**, **dict\_values** й **dict\_items** відповідно, які, по суті, є переліченими змінними і призначені для організації різноманітних циклів при оброблені словників. Зміст цих методів демонструє наступний простий приклад:

```

>>> rainbow = {'red': 620, 'orangr': 585, 'yellow': 575,
 'green': 510, 'skyblue': 480, 'blue': 440, 'purple': 390}
>>> for j in rainbow.items():
 print(j, end='\n')

('red', 620)
('orangr', 585)
('yellow', 575)
('green', 510)
('skyblue', 480)
('blue', 440)
('purple', 390)
>>> for j in rainbow.keys():
 print(j, end='\n')

red
orangr
yellow
green
skyblue
blue
purple
>>> for j in rainbow.values():
 print(j, end='\n')

620
585
575
510
480
440
390

```

Підкреслимо, результат застосування цих методів є перелічена множина значень, оскільки її можна використовувати для організації циклів.

Але це не список, рядок або кортеж, тобто не індексована змінна, а спеціальний тип даних

```

>>> a = rainbow.items[0]
Traceback (most recent call last):
 File "<pyshell#23>", line 1, in <module>
 a = rainbow.items[0]
TypeError: 'builtin_function_or_method' object is not subscriptable
>>> type(rainbow.values())
<class 'dict_values'>
>>> type(rainbow.keys())
<class 'dict_keys'>
>>> type(rainbow.items())
<class 'dict_items'>

```

Але, ця проблема не створює труднощів, оскільки існують вже знайомі функції перетворення типів

```
>>> a = list(rainbow.items())
>>> a
[('red', 620), ('orangr', 585), ('yellow', 575), ('green', 510), ('skyblue', 480),
('blue', 440), ('purple', 390)]
>>> type(a[0])
<class 'tuple'>
>>> b = tuple(rainbow.items())
>>> b
(('red', 620), ('orangr', 585), ('yellow', 575), ('green', 510), ('skyblue', 480),
('blue', 440), ('purple', 390))
```

Із отриманими об'єктами можна діяти відповідно їх нового типу

```
>>> type(a)
<class 'list'>
>>> a.insert(2, ('yellow-green', 540))
>>> a
[('red', 620), ('orangr', 585), ('yellow-green', 540), ('yellow', 575),
('green', 510), ('skyblue', 480), ('blue', 440), ('purple', 390)]
```

### **Метод `pop()`**

І все ж таки, відчуття задоволення не виникає, оскільки обернене перетворення списку у словник неможливе

```
>>> c = dict_items(a)
Traceback (most recent call last):
 File "<pyshell#38>", line 1, in <module>
 c = dict_items(a)
NameError: name 'dict_items' is not defined
```

Як же змінити словник?

Імена ключів можна змінювати методом `pop()`. Загальний формат заміни ключа такий

name dictionary [new key] = name dictionary. pop (old key):

Приклад:

```
>>> rainbow['dark-blue'] = rainbow.pop('blue')
>>> rainbow
{'red': 620, 'orangr': 585, 'yellow': 575, 'green': 510, 'skyblue': 480,
'purple': 390, 'dark-blue': 440}
```

### **Перевірка існування ключа або значення у словнику**

Операції `in` й `not in` дають змогу перевірити, чи існує у списку дане значення. Ці ж операції можна використовувати і для перевірки, чи міститься у словнику певний ключ або значення:

```
>>> rainbow = {'red': 620, 'orange': 585, 'yellow': 575, 'green': 510,
 'skyblue': 480, 'sky': 440, 'purplr': 390}
>>> 'red' in rainbow.keys()
True
>>> 620 in rainbow.values()
True
>>> 'red' in rainbow
True
>>> 620 in rainbow
False
```

Останніх два рядки демонструють, що для ключа проходить й скорочена форма виразу, а для значення – ні.

### *Метод get()*

Перевіряти кожного разу при доступі до ключа, чи є він у словнику було б неправильно. Тому для словників передбачений метод **get()**, який приймає два аргументи: ключ потрібного значення і значення, яке за замовчанням повертається у випадку відсутності ключа у словнику:

```
>>> 'The lenght wave of first rainbow color is ' + \
 str(rainbow.get('red', 0))
'The lenght wave of first rainbow color is 620'
>>> 'The lenght wave of first rainbow color is ' + \
 str(rainbow.get('dark red', 0))
'The lenght wave of first rainbow color is 0'
>>> rainbow['red']
620
>>> rainbow['dark red']
Traceback (most recent call last):
 File "<pyshell#15>", line 1, in <module>
 rainbow['dark red']
KeyError: 'dark red'
```

Важливим є те, що при цьому не виникає помилки і програма продовжує працювати. Також видно, що спроба отримати значення безпосередньо за ключом, який не існує у словнику, веде до помилки.

### *Метод setdefault()*

Досить часто створюється словник, у якому визначені не всі необхідні ключі та їх значення. Розширити, необхідності, словник можна методом **setdefault()**. Це можна зробити за допомогою методу **setdefault()**. Даний метод приймає два аргументи. Перший – ключ, для якого виконується перевірка на присутність у словнику. Другий – значення цього ключа.

Якщо ключ відсутній у словнику, то словник доповнюється цією парою. Якщо ключ присутній у словнику, то метод **setdefault()** повертає актуальне значення цього ключа, а другий аргумент ігнорується.

Приклад:

```
>>> new_rainbow = rainbow.copy()
>>> new_rainbow.setdefault('dark red', 700)
700
>>> new_rainbow
{'red': 620, 'grange': 585, 'yellow': 570, 'green': 510,
'skyblue': 480, 'blue': 440, 'purple': 390, 'dark red':
700}
>>> new_rainbow.setdefault('red', 1000)
620
```

Перший рядок демонструє метод **copy()**, який здійснює неглибоке (див. вище) копіювання словника. Далі, перший виклик методу **setdefault()** змінює словник. Метод **setdefault()** повертає значення нового ключа '**dark red**'. Далі спроба визначити пару '**red**: **700**'. Проте такий ключ існує у словнику. Метод нагадує про це виведенням його актуального значення.

Наступний приклад демонструє застосування цього методу. Наведено код, який у заданому рядку підраховує кількість входженьожної букви.

## Контрольне питання

Цей код може бути використаний у реальних випадках, коли, наприклад, треба дослідити текст з великою кількістю рядків (стаття, книга і т.і.).

Отже:

Як працює цей код? Описати алгоритм.

```
message = 'It was a bright cold day in March, and the clocks \
were striking twelve'
count = {}
for char in message:
 count.setdefault(char, 0)
 count[char] = count[char] + 1

print(message)
print(count)
It was a bright cold day in March, and the clocks were striking twelve
{'I': 1, 't': 5, ' ': 13, 'w': 3, 'a': 5, 's': 3, 'b': 1, 'r': 4, 'i': 4, 'g': 2,
'h': 3, 'c': 4, 'o': 2, 'l': 3, 'd': 3, 'y': 1, 'n': 3, 'M': 1, ',': 1, 'e': 5,
',': 2, 'v': 1}
```

## Форматування друку словників

У прикладах вище словники виводяться на екран у вигляді рядків. Це не завжди зручно для візуального аналізу результатів.

Для форматування друку у Python є модуль **pprint**. Зокрема, для друку словників можна використовувати функцію з такою ж назвою  **pprint()**.

Словники **count** й **rainbow** із попередніх прикладів за допомогою цієї функції можна роздрукувати наступним чином

```
>>> import pprint
>>> pprint.pprint(count)
{' ': 13,
 ',': 1,
 'I': 1,
 'M': 1,
 'a': 5,
 'b': 1,
 'c': 4,
 'd': 3,
 'e': 5,
 'g': 2,
 'h': 3,
 'i': 4,
 'k': 2,
 'l': 3,
 'n': 3,
 'o': 2,
 'r': 4,
 's': 3,
 't': 5,
 'v': 1,
 'w': 3,
 'y': 1}

>>> pprint.pprint(rainbow)
{'green': 510,
 'orange': 585,
 'purplr': 390,
 'red': 620,
 'sky': 440,
 'skyblue': 480,
 'yellow': 575}
>>> print(rainbow)
{'red': 620, 'orange': 585, 'yellow': 575, 'green': 510,
 'skyblue': 480, 'sky': 440, 'purplr': 390}
```

### **Примітка**

1. Зрозуміло, що наведених методів для конструктивної реалізації проектів. У таблиці всі методи для роботи зі словниками та їх призначення. Це дасть змогу знайти на інтернет ресурсах необхідну інформацію.
2. Створити словник можна різними способами, зокрема:
  - 2.1. Інтерактивним введенням, як словник **rainbow** у прикладі вище.
  - 2.2. За допомогою функції **dict**, яка, фактично, перетворює списки або кортежі пар на словник:

```

>>> s = [['grandpa', 'Bill'], ['grandma', 'Sara'], ['dad', 'Tom'], ['mom', 'Ceci']]
>>> dic_s = dict(s)
>>> dic_s
{'grandpa': 'Bill', 'grandma': 'Sara', 'dad': 'Tom', 'mom': 'Ceci'}
>>> t = (('grandpa', 'Bill'), ('grandma', 'Sara'), ('dad', 'Tom'), ('mom', 'Ceci'))
>>> dic_tup= dict(t)
>>> dic_tup
{'grandpa': 'Bill', 'grandma': 'Sara', 'dad': 'Tom', 'mom': 'Ceci'}

```

- 2.3. За допомогою методу **fromkeys**, який перетворює заданий список або кортеж на словник із відповідними ключами і порожніми значеннями цих ключів:

```

>>> parents = ['grandpa', 'grandma', 'dad', 'mom']
>>> dic_par = dict.fromkeys(parents)
>>> dic_par
{'grandpa': None, 'grandma': None, 'dad': None, 'mom': None}

```

- 2.4. За допомогою генераторів словників, аналогічних генераторам списків:

```

>>> parents = ['grandpa', 'grandma', 'dad', 'mom']
>>> names = ['Bill', 'Sara', 'Tom', 'Ceci']
>>> dic_par = { parents[i]: names[i] for i in range(len(parents)) }
>>> dic_par
{'grandpa': 'Bill', 'grandma': 'Sara', 'dad': 'Tom', 'mom': 'Ceci'}

```

| Таблиця Метод                              | Що робить                                                                                                                                                                                |
|--------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>dict.clear()</b>                        | Очищає словник                                                                                                                                                                           |
| <del><b>dict.copy()</b>, <b>deep</b></del> | <del>Повертає копію словника <b>copy</b></del>                                                                                                                                           |
| <b>dict.fromkeys(seq[, value])</b>         | Створює словник з ключами із <b>seq</b> і <b>value</b> )<br>значеннями <b>value</b> (за замовчуванням                                                                                    |
| <b>dict.get(key[, default])</b>            | None).                                                                                                                                                                                   |
| <b>dict.items()</b>                        | Повертає пари (ключ, значення) <b>dict.keys()</b> -                                                                                                                                      |
| <b>dict.pop(key[, default])</b>            | Повертає ключі у словнику <b>dict.pop(key[, default])</b> Видаляє ключ і<br>повертає його значення. Якщо ключа немає,<br>повертає default]) чення. Якщо ключа немає,                     |
| <b>dict.popitem()</b>                      | default (за замовчуванням видає помилку) Видаляє<br>і повертає пару (ключ, зна-<br>чення). Якщо словник порожній, видає помилку                                                          |
| <b>dict.setdefault(key[, default])</b>     | KeyError                                                                                                                                                                                 |
| <b>dict.update([other])</b>                | Повертає значення ключа, якщо його<br>(за замовчуванням немає, створює ключ із значенням<br>default None).<br>Обновлює словник, додаючи пари<br>(ключ, значення) із other. Ключі, що іс- |

|                        |                                     |          |
|------------------------|-------------------------------------|----------|
| -                      | нують, перезаписуються.             | Повертає |
|                        | None, якщо словник не змінився      |          |
| <b>dict.values()</b> - | Повертає значення ключів у словнику |          |

## ***Резюме***

Списки і словник є одно- або багаторівневими ієархіями даних різноманітних типів. Доступ до структурних частин цих об'єктів здійснюється за допомогою індексів аналогічно, з точки зору синтаксису виразів.

Принципова відміна полягає у тому, що списки – упорядковані дані і доступ здійснюється за допомогою числових індексів. У словниках індексація здійснюється за допомогою ключів, причому ключом можуть бути дані будь-якого базового типу

```
>>> a = {True: None, 1: False}
```

Завдяки цьому, словники є ідеальною структурою даних для подання реальних об'єктів на підставі фреймової моделі при вирішення задач машинного навчання (machine learning) та інтелектуального аналізу даних (data science)

Розглянутого матеріалу цілком достатньо для розроблення корисних програм при розв'язуванні й інших важливих задач розроблення й завантаження вебсторінок, оброблення електронних таблиць, відправляти текстові повідомлення і т. і.

## **Контрольні питання**

1. Як виглядає код для порожнього словника?
2. Як виглядає код доступу до ключа словника і до значення ключа? Навести приклад.
3. У чому збіжність і принципова відміна між словником і списком, як структур даних?
4. Що таке фреймова модель реальних об'єктів?
5. Чому, на вашу думку, словник мови Python ідеальна структура даних для програмного подання реальних об'єктів.
6. Як створити словник?
7. Як перевірити, чи містить словник заданий ключ, або значення ключа?
8. Як додати перевірити, чи містить словник заданий ключ і, якщо ні, то додати цей ключ і його значення
9. Які модуль і функцію варто використовувати для форматування виведення словників на екран і чому?

## **Навчальний проект (курсова робота, частина 2)**

## ***Інвентар гри***

Ви створюєте відео гру. Гравець має рюкзак, у якому зберігає свій інвентар. Наприклад, у грі «Сталкер» у рюкзаку зберігаються декілька видів зброї, ліки і т.д.

Напишіть функції, які:

1. Інтерактивно формує вміст рюкзака (за вашим вподобаннями).
2. Виводить на екран вміст рюкзака.
3. Дає змогу змінювати кількість предметів у рюкзаку.
4. Додавати нові види інвентарю.
5. Ввести обмеження на загальну кількість різновидів речей у рюкзаку.
6. Реалізувати можливість викидати непотрібні речі, зменшуючи їх кількість, і повністю.

Наприклад, вміст рюкзака у грі Сталкер:

Inventory: AK 12: 1 Chezer: 2

first aid kit: 3

vodka “Kazaki”: 3 Total

number of items: 4

Total number of values: 9

## Лабораторна робота № 12

### Тема: «Препроцесінг або оброблення сиріх даних»

*Мета:*

*Отримати погляд на сирі дані та поширені методи препроцесінгу*

### Теоретичний мінімум

#### Сирі дані

На практиці даним, отриманим від замовника, дуже часто притаманні ряд незручних для виконавця властивостей – величезний об’єм, величезний розмах (діапазон), неповнота, некоректність, нечислові значення ознак та ін. У наслідок цього попереднє оброблення даних (**препроцесінг**) є дуже важливим етапом машинного навчання.

Загальних рекомендацій у цьому питанні не існує. Проте, на сьогодні розроблено достатньо модулів, які містять різноманітний інструментарій і виконавець має можливість вибирати необхідне у багатьох конкретних випадках.

Далі розглядається декілька простих, але досить поширених методів препроцесінгу

- бінаризація;
- вилучення середнього;
- масштабування; - нормалізація;
- кодування міток.

Відповідні функції містяться у модулі **preprocessing** бібліотеки **scikit-learn**. Реалізація алгоритмів машинного навчання передбачає, що перед початком навчання дані вже відформовані певним чином. Поширеним форматом даних для алгоритмів машинного навчання є масиви створені методом **array** бібліотеки **Numpy**, яка містить і методи оброблення даних такого типу.

```
>>> import numpy as np
>>> from sklearn import preprocessing
```

Для демонстрації прикладів застосування препроцесінгу створимо штучні дані такого формату

```
>>> x = np.array([[5.1, -2.9, 3.3],
 [-1.2, 7.8, 3.3],
 [3.9, 0.4, 2.1],
 [7.3,-9.9,-4.5]])
```

## Препроцесінг Бінаризація

Цей процес застосовується у тих випадках, коли треба перетворити числові дані у булеві. Наприклад, якщо елемент масиву більше певного порогового значення, він заміняється на 1. Якщо навпаки – на 0.

Порогове значення у прикладі – середнє арифметичне елементів масиву

```
>>> ave = np.mean(x)
>>> data_binarized = preprocessing.Binarizer(threshold=ave).\\
 transform(x)
>>> print('\nBinarized data (mean = ', ave, ') :\n', data_binarized)

Binarized data (mean = 1.2249999999999996) :
[[1. 0. 1.]
[0. 1. 1.]
[1. 0. 1.]
[1. 0. 0.]]
```

**!Функція mean() обчислює середнє всіх елементів масиву**

У даному прикладі зміст бінаризації полягає у присвоюванні мітки «1» елементам більшим за середнє арифметичне і «0» меншим за нього. Код цілком зрозумілий і не потребує коментарів.

На практиці може зустрінутися подібна задача, але ускладнена конкретними умовами. Тоді розроблення необхідної функції – це завдання виконавця.

## **Завдання 1**

Для масиву  $x$  (див. вище) виконати наступне:

1. Знайти середнє відхилення  $div$  значень масиву від середнього арифметичного значення.

2. Розробити функцію, яка присвоює мітки елементам масиву за наступним правилом:

$$label = \begin{cases} 0 & if \ x[i] > ave + div \\ 1 & if \ ave - div \leq x[i] \leq ave + div \\ 0 & if \ x[i] < ave - div \end{cases}$$

3. Виконати перетворення для масиву із прикладу вище.

### Вилучення середнього

Вилучення середнього – методика попереднього оброблення даних, часто використовується у машинному навчанні. Після такого оброблення значення ознак (**feature**) будуть центруватися навколо нуля, що часто **спрощує** подальші обчислення.

Для масиву із попереднього прикладу обчислимо середнє у кожному стовпчику масиву і стандарте відхилення елементів кожного стовпчика від свого середнього. Потім вилучимо середнє значення і знову обчислимо ті ж величини.

```

import numpy as np
from sklearn import preprocessing
x = np.array([[5.1, -2.9, 3.3],
 [-1.2, 7.8, -6.1],
 [3.9, 0.4, 2.1],
 [7.3, -9.9, -4.5]])
print('For input data: ', '\n x = ', x,
 '\n mean = ', x.mean(axis=0), '\n',
 'Std deviation = ', x.std(axis=0))
print('\n After mean scale: ', '\n x_scaled = ',
 preprocessing.scale(x),
 '\n mean_scaled = ',
 (preprocessing.scale(x)).mean(axis=0),
 '\n Std deviation_scaled = ',
 (preprocessing.scale(x)).std(axis=0))

print('\n sum array elements: ',
 '\n Befor sum = ', sum(x),
 '\n After sum = ', sum(preprocessing.scale(x)))
For input data:
x = [[5.1 -2.9 3.3]
[-1.2 7.8 -6.1]
[3.9 0.4 2.1]
[7.3 -9.9 -4.5]]
mean = [3.775 -1.15 -1.3]
Std deviation = [3.12039661 6.36651396 4.0620192]

After mean scale:
x_scaled = [[0.42462551 -0.2748757 1.13244172]
[-1.59434861 1.40579288 -1.18167831]
[0.04005901 0.24346134 0.83702214]
[1.12966409 -1.37437851 -0.78778554]]
mean_scaled = [1.11022302e-16 0.00000000e+00 2.77555756e-17]
Std deviation_scaled = [1. 1. 1.]

sum array elements:
Befor sum = [15.1 -4.6 -5.2]
After sum = [4.44089210e-16 0.00000000e+00 1.11022302e-16]

```

**!Власний метод .mean() масиву обчислює середнє для кожного рядка масиву**

Видно, що після вилучення середнього дані у кожному стовпчику центруються навколо нуля і їх сума стає рівною нулю.

Прикладом полегшення розв'язування, за умови централізації даних навчальної вибірки, тобто значень ознак і значень цільової функції, є знаходження функції лінійної регресії. Формули для коефіцієнтів функції містять такі суми і формули суттєво спрощуються.

Якщо навчальна вибірка має великий обсяг і (або) вектори містять багато елементів, використання центрованих даних якісно підвищує ефективність знаходження лінійної регресії.

## **Завдання 2\*\* (до «відмінно» на екзамені):**

Знайти у джерелах формули обчислення коефіцієнтів лінійної регресії у одновимірному випадку (об'єкт має одну ознаку) і перетворити їх для випадку даних з вилученим середнім.

### **Масштабування**

У векторі ознак значення можуть змінюватися у великих границях (великий розмах даних) і (або), наперед невідомих, наприклад, випадкових, границях. Невизначеність алгоритмізації у таких випадках може бути усунена за рахунок масштабування даних і приведення їх до наперед визначеного проміжку значень.

У прикладі наведена саме така ситуація. Об'єкти мають одну ознаку. У вибірці п'ять об'єктів. Межи, у яких може змінюватися ознака, є випадковими і її значення у цих межах також випадкові. За рахунок масштабування ознака приводиться до проміжку [0, 1] незалежно від того, як будуть змінюватися значення ознаки.

```
import numpy as np
import random
from sklearn import preprocessing

a = random.uniform(-10, 0)
b = random.uniform(2, 10)
array must be column !
x = np.array([[(b-a)*np.random.random() + a] for i in range(5)])
print(' Random values of objects feature : \n', x)

data_scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))

data_scaled = data_scaler.fit_transform(x)
print('\n Scaled values of objects feature : \n', data_scaled)

Random values of objects feature :
[[-0.92314103]
 [8.12227376]
 [1.73913879]
 [-1.80939601]
 [5.17039543]]

Scaled values of objects feature :
[[0.08923524]
 [1.]
 [0.35729488]
 [0.]
 [0.70278127]]
```

## Нормалізація даних

Сирі дані є результатами вимірювання, на що впливає багато факторів. Процес нормалізації полягає у згладжуванні значень однакових ознак (вздовж рядків матриці ознак) таким чином, щоб їх можна було б порівнювати.

Одним з методів є нормалізація – приведення даних до деякої загальної шкали.

У машинному навчанні використовуються різні форми нормалізації.

**L1-нормалізація** (метод найменших абсолютнох відхилень (Least Absolute Deviations)). Алгоритм полягає у знаходженні суми абсолютнох значень елементів рядка, обчисленні частки кожного елемента рядка і цієї суми, заміні кожного елементу рядка на відповідну частку. Алгоритм виконується для кожного рядка матриці ознак. Загальність підходу проявляється у тому, що сума абсолютнох значень елементів вздовж кожного рядка рівна 1 . Наведений нижче код реалізує цей алгоритм

```
x = np.array([[5.1,-2.9, 3.3],
 [-1.2, 7.8,-6.1],
 [3.9, 0.4, 2.1],
 [7.3,-9.9,-4.5]])
x_normal = [x[:,0]/sum(abs(x[:,0])),
 x[:,1]/sum(abs(x[:,1])),
 x[:,2]/sum(abs(x[:,2])),
 x[:,3]/sum(abs(x[:,3]))]

print("\nL1 normalized data:\n")
for i in x_normal:
 print(x_normal[i], '\n')

L1 normalized data:
[0.45132743 -0.25663717 0.2920354]
[0.45132743 -0.25663717 0.2920354]
[0.45132743 -0.25663717 0.2920354]
[0.45132743 -0.25663717 0.2920354]
```

**L2-нормалізація** використовує метод найменших квадратів, що забезпечує рівність 1 суми квадратів значень вздовж кожного рядка.

Техніка L1-нормалізації вважається більш надійною порівняно із L2-нормалізацією, оскільки вона менш чутлива до викидів (**чому?**).

Якщо викиди у даних треба враховувати, кращим вибором є L2-нормалізація.

Модуль **preprocessing** містить відповідні методи. Виконаємо ці перетворення

```

import numpy as np
from sklearn import preprocessing

x = np.array([[5.1,-2.9, 3.3],
 [-1.2, 7.8,-6.1],
 [3.9, 0.4, 2.1],
 [7.3,-9.9,-4.5]])

x_normal_L1 = preprocessing.normalize(x, norm = 'l1')
x_normal_L2 = preprocessing.normalize(x, norm = 'l2')
print('\nL1 normalized data:\n', x_normal_L1)
print('\nL2 normalized data:\n', x_normal_L2)

L1 normalized data:
[[0.45132743 -0.25663717 0.2920354]
 [-0.0794702 0.51655629 -0.40397351]
 [0.609375 0.0625 0.328125]
 [0.33640553 -0.4562212 -0.20737327]]

L2 normalized data:
[[0.75765788 -0.43082507 0.49024922]
 [-0.12030718 0.78199664 -0.61156148]
 [0.87690281 0.08993875 0.47217844]
 [0.55734935 -0.75585734 -0.34357152]]

```

Результати L1 нормалізації, зрозуміло, співпадають з наведеними у попередньому прикладі.

## Кодування міток

У задачах класифікації дані маркуються певними мітками (labels). Мітками можуть бути слова, числа або інші об'єкти.

На практиці, як правило, мітками класів даних є слова. Це природно для людини, яка готує дані. Функції машинного навчання, які містить бібліотека sklearn, передбачають, що мітки є числами, оскільки це природно для комп'ютера. Для перетворення слів у числа застосовується **кодування міток**. Під кодуванням міток (**label encoding**) розуміється процес встановлення однозначної відповідності між списком міток і списком цілих чисел. Необхідно передбачити й обернене перетворення.

## Приклад

```
import numpy as np
from sklearn import preprocessing

#code labels
input_labels = ['red', 'black', 'red', 'green', \
 'black', 'yellow', 'white', 'red']
encoder = preprocessing.LabelEncoder()
encoder.fit(input_labels)

print('Coding***', '\nLabel mapping:')
for i, item in enumerate(encoder.classes_):
 print(item, '-->', i)

testing
test_labels = ['green', 'red', 'black']
encoded_values = encoder.transform(test_labels)
print('\nTesting***', '\nLabels = ', test_labels)
print('Encoded values =', list(encoded_values))

#decode numers
encoded_values = [3, 0, 4, 1]
decoded_list = encoder.inverse_transform(encoded_values)
print('\nDecoding***', '\nEncoded values =', encoded_values)
print('Decoded labels =', list(decoded_list))

Coding***
Label mapping:
black --> 0
green --> 1
red --> 2
white --> 3
yellow --> 4

Testing***
Labels = ['green', 'red', 'black']
Encoded values = [1, 2, 0]

Decoding***
Encoded values = [3, 0, 4, 1]
Decoded labels = ['white', 'black', 'yellow', 'green']
```

## Завдання 3

Замовник передав файл *Salary.xlsx* (див. Додаток) із даними про середню заробітну плату у регіонах України за кілька років. Виконати наступне:

1. Завантажити дані за роки (запитати викладача) і конвертувати їх у масив **Numpy** у якому кількість стовпчиків відповідає кількості років спостереження, кількість рядків – кількості регіонів України.
2. Виконати бінаризацію даних. Як можна інтерпретувати бінаризовані дані?
3. Виконати масштабування даних. У чому доцільність такого оброблення?
4. Виконати нормалізацію даних. У чому доцільність такого оброблення?
5. Виконати кодування міток (назва регіону).

6. Візуалізувати результати.

**Примітка:**

Достатній інструментарій для роботи з *Excel* можна знайти у лабораторній роботі № 3.

## Лабораторна робота № 13

### Тема: «Задача класифікації»

**Мета:**

За наданим зразком і даними виконати розв'язання типової задачі класифікації

#### Теоретичний мінімум

Основні етапи розв'язування задачі класифікації такі:

**Етап 1. Завантаження і підготовка даних:**

Дані, як правило, створюються **Замовником**, що обумовлює їх зміст і структуру.

1.1. При підготовці даних ключову роль може відігравати представник **Замовника**, так званий **«owner»** проекту, який, з одного боку, володіє предметною областю та інтересами замовника, а з іншого, має необхідні знання та навички щодо організації інтерфейсу із **Виконавцем**.

1.2. Типовим є випадок, коли від замовника можна отримати тільки файл з даними. Цей файл містить дані про предметну область і об'єкти класифікації. Проте інформацію щодо реалізації задачі класифікації приходиться отримувати самостійно, шляхом аналізу цього файлу.

1.3. Формат файлу також може бути самий різний: .xlsx, .txt, .csv, .html, .docx та інші. Бібліотеки **scikit-learn** та **pandas** мають необхідні функції для завантаження таких даних і перетворення їх у необхідні об'єкти.

**Етап 2. Візуалізація і аналіз даних.**

2.1. Перед тим, як будувати модель машинного навчання, варто спробувати візуально проаналізувати дані, щоб зрозуміти, чи можливо легко розв'язати поставлену задачу класифікації без машинного навчання. Як ні, то чи містить наданий файл всю інформацію, достатню для розв'язування цієї задачі методами машинного навчання. Зокрема, чи взагалі можна класифікувати надані об'єкти.

2.2. Візуалізація даних – це, у простих випадках, надійний спосіб виявити викиди (нетиповість) та інші аномалії у даних. Наприклад, поширеним є випадок, коли однакові ознаки різних об'єктів можуть бути надані у різних

системах вимірювання. Наприклад, довжина – метри, фути або сантиметри і т.п.

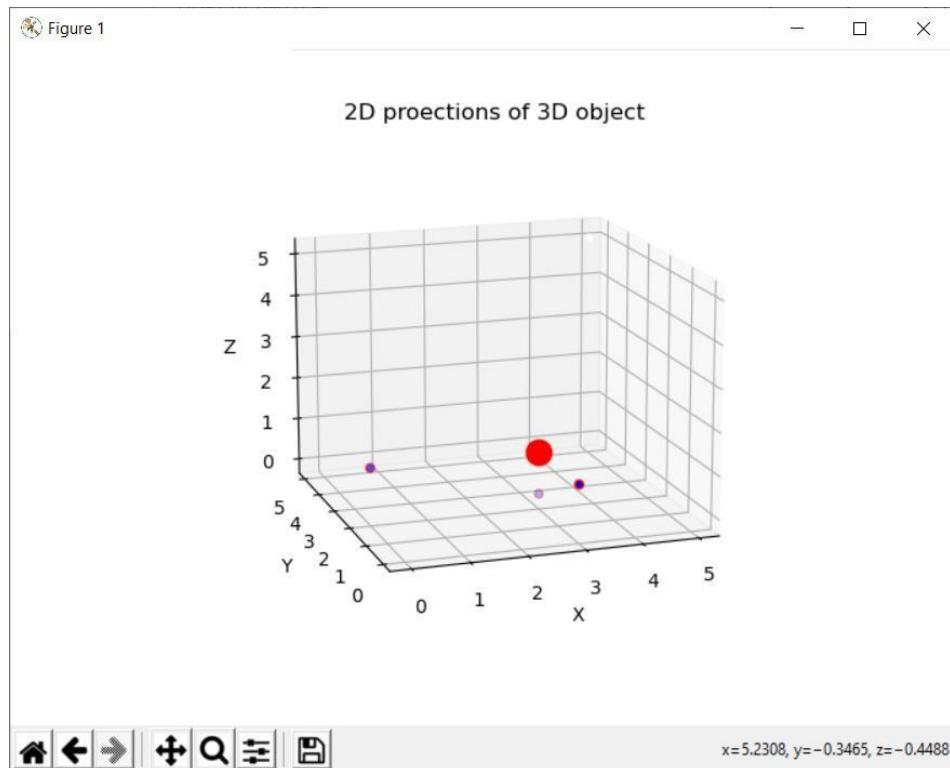
2.3. Значенню кожної ознаки завжди можна поставити у відповідність дійсне або ціле число і тоді візуальним образом об'єкту є **точка** у певному  $n$ -вимірному просторі, де  $n$  – кількість ознак. На рисунку наведений код геометричне подання Проте, на екрані комп’ютера можна зобразити тільки точку на площині. Поширеним виходом з положення є подання образу об'єкту з  $n$ -ознаками, як сукупність  $C^2_n$  плоских картинок – координатних площин, з точки зору  $n$ -вимірної геометрії. Далі наведений код та З точки зору машинного навчання, кожна картинка є прецедентом одночасної появи значень пари ознак з  $C^2_n$  можливих пар для даного об'єкту. Отже, таке подання є повним.

Навчальній вибірці із  $l$  об'єктів буле вілповілати  $C^2$  плоских картинок. на кожній з яких зображені  $l$  точок. І така картинка є преелентом одночасної появи пан значень. Якщо кожному класу поставити в вілповільність окремий колір, по характеру групування точок одного кольору можна робити висновки, щодо існування розв'язку задачі класифікації.

```
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
import numpy as np
x = np.array([3, 3, 0])
y = np.array([2, 0, 2])
z = np.array([0, 1, 1])

fig = plt.figure(figsize = (7, 5))
ax = plt.axes(projection = '3d')
ax.scatter3D(x, y, z, c = 'blue',
 s = 20, edgecolor = 'red')
ax.scatter3D([3], [2], [1], c = 'red', s = 180)
ax.scatter3D([5], [5], [5], c = 'white', s = 10)

plt.title('2D proections of 3D object')
|
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.show()
```



```

import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize = [10, 10])
plt.xticks([])
plt.yticks([])

plt.title('Image of 3D of an object using\\
matrix of two-dimensional projections')

ax2 = fig.add_subplot(3, 3, 2)
ax2.scatter([0, 2, 5], [0, 3, 5], c = ['white', 'red', 'white'],
 s = 60, edgecolors= ['white' , 'green', 'white'])
plt.xlabel('X')
plt.ylabel('Y')

ax3 = fig.add_subplot(3, 3, 3)
ax3.scatter([0, 2, 5], [0, 1, 5], c = ['white', 'red', 'white'],
 s = 60, edgecolors= ['white' , 'green', 'white'])
plt.xlabel('X')
plt.ylabel('Z')

ax4 = fig.add_subplot(3, 3, 4)
ax4.scatter([0, 3, 5], [0, 2, 5], c = ['white', 'red', 'white'],
 s = 60, edgecolors= ['white' , 'green', 'white'])
plt.xlabel('Y')
plt.ylabel('X')

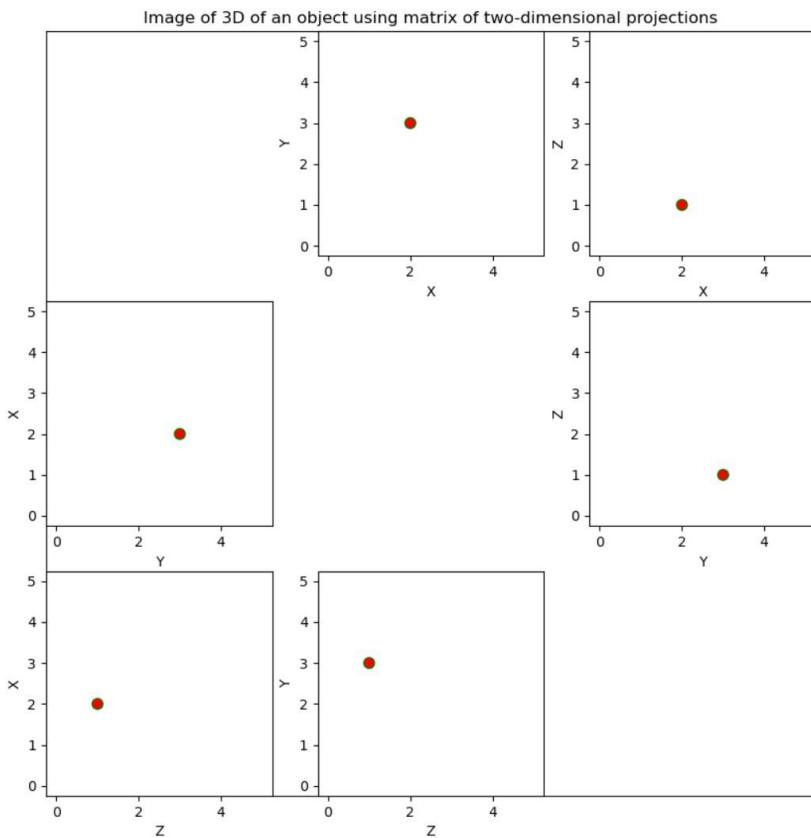
ax6 = fig.add_subplot(3, 3, 6)
ax6.scatter([0, 3, 5], [0, 1, 5], c = ['white', 'red', 'white'],
 s = 60, edgecolors= ['white' , 'green', 'white'])
plt.xlabel('Y')
plt.ylabel('Z')

ax7 = fig.add_subplot(3, 3, 7)
ax7.scatter([0, 1, 5], [0, 2, 5], c = ['white', 'red', 'white'],
 s = 60, edgecolors= ['white' , 'green', 'white'])
plt.xlabel('Z')
plt.ylabel('X')

ax8 = fig.add_subplot(3, 3, 8)
ax8.scatter([0, 1, 5], [0, 3, 5], c = ['white', 'red', 'white'],
 s = 60, edgecolors= ['white' , 'green', 'white'])
plt.xlabel('Z')
plt.ylabel('Y')

plt.show()

```



Множину значення ознаки для всієї вибірки можна уявляти, як значення випадковою величини. Гістограма відносних частот є ще одним маркером, що вказує на існування класів у навчальній вибірці. Зрозуміло, якщо форма гістограми над проміжком значень ознаки наближається до прямокутника, розподіл значень близький до рівномірного і говорить про існування класів недоцільно.

З іншого боку, якщо гістограма має явно виражені пики, то за кількістю піків можна висказати припущення про кількість класів об'єктів у вибірці. Зауважимо, що на сьогодні розроблено багато різних алгоритмів класифікації і для навчання деяких з них необхідно виказати припущення про кількість класів у вибірці. У прикладі із ірисами, гістограми розташовані по діагоналі матриці, яка відображає парні прецеденти що до ознак ірисів.

### *Eтап 3. Вибір і навчання моделі. Оцінка й оптимізація її якості*

У даній роботі вибір моделі навчання за викладачем. Пропонується модель kmeans із одним найближчим сусідом. Вибір моделі у більш загальних випадках – матеріал для самостійної роботи.

### **Завдання:**

У Додатку дано посилання на файл з теорією щодо задачі класифікації і розглянутий випадок, орієнтований на п. 1.2. Тобто у даній роботі із змістом файлу необхідно розбиратися самостійно.

1. Поновити код і результати про класифікацію квіток ірису.
2. Діючи за аналогією, розв'язати задачу класифікації заданих об'єктів:
  - 2.1. За допомогою функції **load\_wine** завантажити із сховища **sklearn** дані.
  - 2.2. Описати ці дані і сформулювати для них задачу класифікації
  - 2.3. Візуалізувати дані. Звернути увагу на якість отриманих картинок, що можна досягнути зміною параметрів функції, що буде відповідні об'єкти.
  - 2.4. За допомогою методу k-means провести навчання моделі класифікації. Змінюючи кількість найближчих сусідів, оптимізувати якість моделі класифікації.

### **Примітка:**

*Зміст і структура початкових даних можуть суттєво відрізнятися від прикладу з ірисом у Додатку.*

## Лабораторна робота № 14

### Тема: «Регресія» *Мета:*

4. Засвоїти різницю між класифікацією і регресією – задачами одного класу, але якісно різними цільовими функціями.
5. Засвоїти поняття недонаавчання та перенавчання алгоритму.
6. Виконати типове завдання з пошуку лінійної регресії

### Теоретичний мінімум

Класифікація і регресія – це прогнозування значення цільової функції на підставі значень вектору ознак. Різниця полягає у природі цільової функції. Якщо цільова функція може приймати тільки скінчену кількість значень (мітки класів), це класифікація. Якщо цільова функція може приймати значення у деякому проміжку дійсних чисел – це регресія.

### Приклад:

На підставі ознак деякої персони необхідно дати прогноз щодо її майнового стану. Якщо цільова функція буде приймати значення «бідний», «середняк» або «заможний», то прогнозування є класифікацією. Якщо необхідно прогнозувати суму річного доходу, то це регресія.

Стратегія навчання в обох випадках одна. Навчальна вибірка ділиться на тренувальний і тестовий набори. Параметри алгоритму підбираються при обробленні тренувального набору, а перевіряється узагальненість моделі на тестовому набору.

Методи навчання також схожі. Метод найближчих  $k$ -сусідів може бути застосований і в задачах класифікації, і в задачах регресії. Метод, побудований на пошуках цільової функції, виходячи із умови мінімуму суми абсолютнох відхилень, або квадратів відхилень, також застосований в обох випадках.

Різниця полягає у тому, що у задачах класифікації цільова функція відшукується у вигляді, так званої, кусково-сталої функції (функція Хевісайда). У якості моделі такої функції часто використовується сігмоїдальна функція (стара знайома за нейронними мережам). У задачах регресії цільова функція є

неперервною функцією неперервних змінних. Часто, моделлю такої функції є поліном.

Досить поширеним у практиці машинного навчання є припущення про лінійну регресію ознак і цільової функції. Тобто, моделлю цільової функції є поліном першого ступеня.

$$y = w_0x_0 + w_1x_1 + \dots + w_nx_n,$$

де  $w_i$  – параметри алгоритму,  $(x_0, x_1, \dots, x_n)$  – вектор ознак досліджуваного об'єкту.

Таке припущення є найпростішим, але часто забезпечує цілком задовільні результати при високій швидкості прогнозування.

Якщо при такому припущені якість прогнозу недостатня, може бути використана більш складна модель цільової функції, у вигляді поліному вищих ступенів. Обчислювальна складність моделі при цьому має нелінійне зростання, але це не суттєвий фактор при сучасних характеристиках обчислювальної техніки. Суттєвою є проблема недонавчання, або перенавчання моделі.

## **Недонавчання й перенавчання моделі**

Одна з центральних проблем математичного навчання – це компроміс між складністю моделі класифікації (регресії) і її узагальненістю. Нагадаємо, модель навчається на тренувальному наборі, перевіряється на тестовому наборі, а мета навчання узагальнення моделі на дані, які не зустрічалися раніше.

Цю проблему можна охарактеризувати двома поняттями – **недонавчання (underfitting)** та **перенавчання (overfitting)** моделі класифікації (регресії).

**Недонавчання** – ситуація, коли зміною параметрів алгоритму не вдається знайти функцію, яка дає задовільні прогнози.

Найбільш поширенна причина недонавчання – коли складність даних вище за складність моделі, яку застосував виконавець. З точки зору алгоритмізації, причиною недонавчання є недостатня кількість вільних параметрів алгоритму у застосованій моделі.

Рішенням такої проблеми, природно, буде ускладнення моделі.

*Приклад.* У лінійній моделі (див. вище), кількість вільних параметрів рівна кількості ознак. Модель наступної складності – поліном другого ступеня. Такий алгоритм буде містити параметрів у двічі більше.

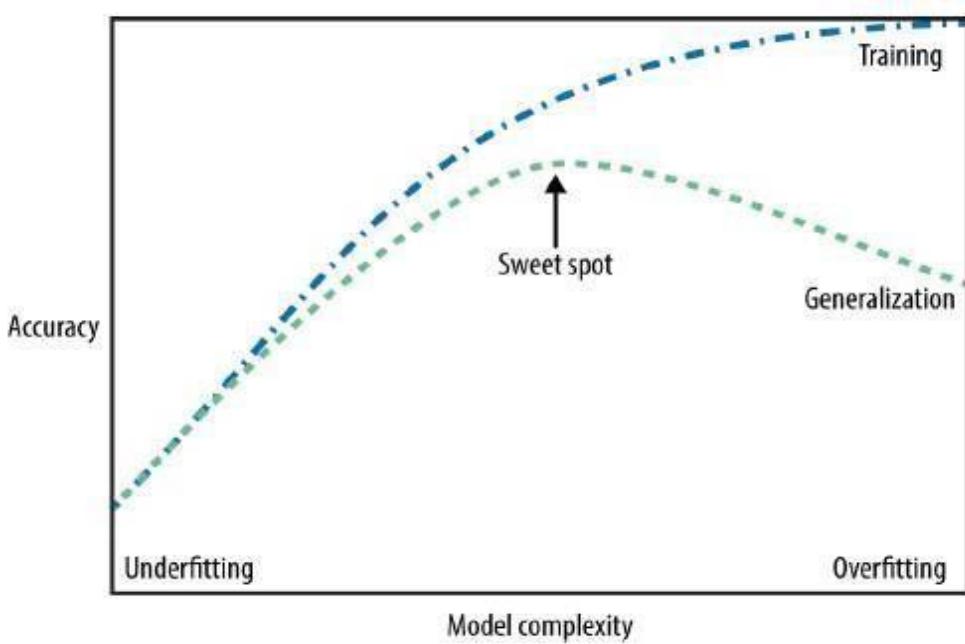
**Перенавчання** – протилежний недонавчанню ефект. Виконавець часто знаходить перед спокусою підвищувати складність моделі, оскільки величина

похиби прогнозу при навчанні на тренувальному наборі будуть зменшуватися. Проте, по закінченню навчання, на тестовому наборі модель видає неприйнятну похибку прогнозу.

З точки зору алгоритмізації, складна модель містить забагато вільних параметрів. При навчанні моделі «ансамбль» параметрів налаштовується на розпізнавання не тільки загальних, а й особистих рис образів тренувального набору. Як кажуть, модель втрачає узагальненість. Природно, на тестових даних, які для моделі є новими, прогноз може бути неприйнятним.

*Приклад.* Якщо при навчанні моделі за методом найближчих  $k$ -сусідів параметр  $k$  встановити рівним кількості об'єктів у тренувальному наборі, похибка прогнозу моделі на тренувальному наборі буде рівна нулю (**чому?**). Разом з цим, для тестового набору прогноз буде неприйнятним.

**Критерієм вдалого навчання** моделі є прийнятна помилка прогнозу, отримана на даних, на яких модель навчалась (тренувальний набір) і відповідно прийнятна помилка прогнозу, отримана навченою моделлю для нових даних, на яких вона не навчалась (тестовий набір). Схематично це показано на рисунку. Проблема оптимального вибору складності моделі, одна з актуальніших у машинному навчанні, не вирішена досі.



**Рис.** Компроміс між складністю та узагальненістю моделі у процесі навчання

Помилка обчислюється, як відхилення значення прогнозу, що дає навчена модель від значень прогнозу, які відомі для тренувального та тестового наборів.

Методики обчислення відхилення можуть бути різні (середнє абсолютне відхилення, середня квадратична похибка тощо). Це визначається аналітиком – одна з ролей у команді виконавці.

## Створення регресору однієї змінної

### *Приклад*

Навчальна вибірка зберігається у файлі **data\_singlevar\_regr.txt**. Фрагмент даних показаний на рисунку. Текстовий формат – поширена форма даних від замовника. Модуль **numpy** містить відповідну функцію для завантаження даних у такому форматі. Дані – двовимірний масив з двома стовпчиками. Перший – значення ознаки. Другий – відповідні значення залежності змінної. На рисунку показаний фрагмент даних з подільником (,).

| data_singlevar_regr – Блокнот |        |
|-------------------------------|--------|
|                               |        |
| Файл                          | Правка |
| -0.86, 4.38                   |        |
| 2.58, 6.97                    |        |
| 4.17, 7.01                    |        |
| 2.6, 5.44                     |        |
| 5.13, 6.45                    |        |
| 3.23, 5.49                    |        |
| -0.26, 4.25                   |        |
| 2.76, 5.94                    |        |
| 0.47, 4.8                     |        |
| -3.9, 2.7                     |        |
| 0.27, 3.26                    |        |
| 2.88, 6.48                    |        |
| -0.54, 4.08                   |        |

Модуль **pickle** містить функції для збереження у вигляді файлу і зчитування з файлу навченої моделі. Модель може мати складну структуру і розміри, що потребую спеціальних алгоритмів.

Модуль **numpy** містить функції для оброблення масивів.

Модуль **sklearn** (це scikit-learn) містить функції й об'єктні моделі задач класифікації. У даному випадку використовуються модулі:

- **linear\_model** – лінійна регресія;
- метод **.metrics** – обчислення похибок прогнозу;
- функція **train\_test\_split**, яка перемішує дані (для чого?), ділить навчальну вибірку на тренувальний і тестовий набори;
- **matplotlib** – візуалізація даних.

```

import pickle

import numpy as np
from sklearn import linear_model
import sklearn.metrics as sm
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

input_file = 'data_singlevar_regr.txt'
data = np.loadtxt(input_file, delimiter = ',')
X, y = data[:, :-1], data[:, -1]

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

regressor = linear_model.LinearRegression()

regressor.fit(X_train, y_train)

y_test_pred = regressor.predict(X_test)

#Visualisation
plt.scatter(X_train, y_train, color = 'red', s = 10)
plt.scatter(X_test, y_test, color = 'green')
Y_pred = regressor.predict(X)

plt.plot(X, Y_pred, color = 'black', linewidth = 2)

plt.xticks(())
plt.yticks(())
plt.xlabel('X')
plt.ylabel('y')

plt.show()

feature metrics

print('Leaner regressor perfomence:')
print('Mean absolute error= ', round(sm.mean_absolute_error(
 y_test, y_test_pred),2))
print('Mean squared error= ', round(sm.mean_squared_error(
 y_test, y_test_pred),2))
print('Median anabsolute error= ', round(sm.median_absolute_error(
 y_test, y_test_pred),2))
print('Explain variance score= ', round(sm.explained_variance_score(
 y_test, y_test_pred),2))
print('R2 score= ', round(sm.r2_score(y_test, y_test_pred),2))

```

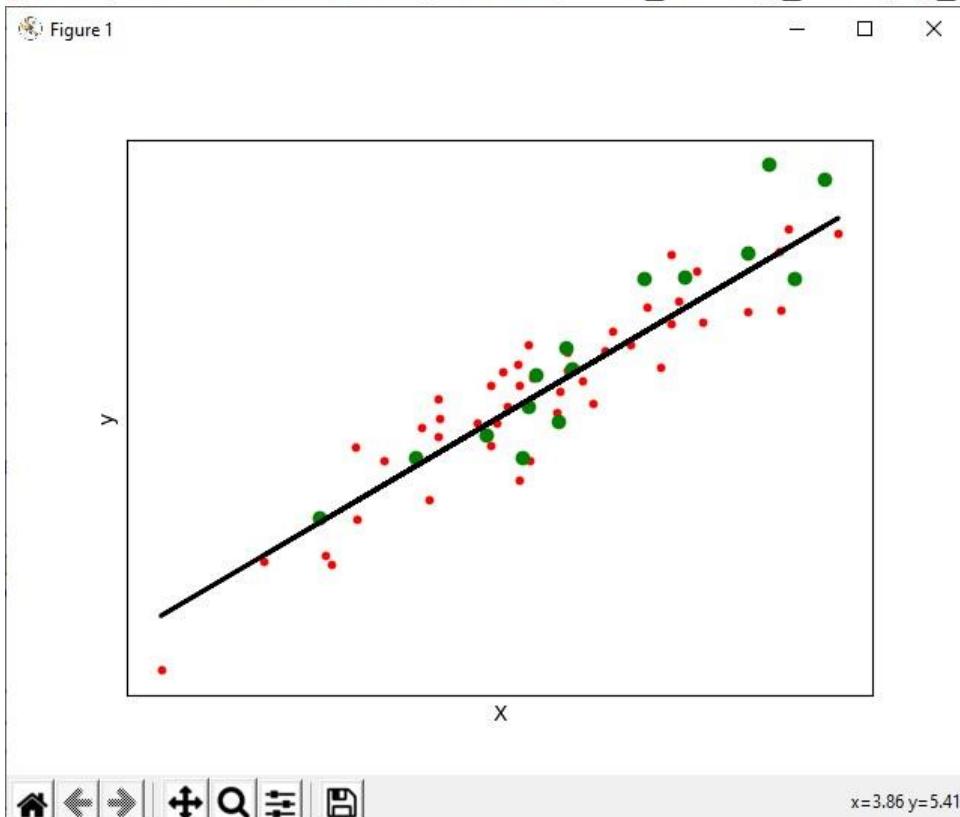
```

#output trained model to file using the module 'pickle'
output_model_file='model.pkl'
with open(output_model_file, 'wb') as f:
 pickle.dump(regressor,f)

#predict for training data
with open(output_model_file, 'rb') as f:
 regressor_model = pickle.load(f)

y_train_pred_new = regressor_model.predict(X_train)
print('\nTrain mean absolute error= ',
 round(sm.mean_absolute_error(y_train,
 y_train_pred_new),2))
print('Train mean squared error= ', round(sm.mean_squared_error(
 y_train, y_train_pred_new),2))
print('Train median absolute error= ', round(sm.median_absolute_error(
 y_train, y_train_pred_new),2))
print('Train explain variance score= ', round(sm.explained_variance_score(
 y_train, y_train_pred_new),2))
print('Train R2 score= ', round(sm.r2_score(y_train, y_train_pred_new),2))

```



На рисунку червоними крапками показаний тренувальний набір, зеленими – тестовий набір, чорною лінією – цільова функція лінійної регресії між ознакою **X** й прогнозом **Y**.

```
Leaner regressor perfomence:
Mean absolute error= 0.68
Mean squared error= 0.75
Median anabsolute error= 0.7
Explain variance score= 0.87
R2 score= 0.85

Train mean absolute error= 0.64
Train mean squared error= 0.6
Train median anabsolute error= 0.55
Train explain variance score= 0.86
Train R2 score= 0.86
```

По закінченню навчання за різними методиками обчислюються похибки (**feature metrics**).

Середнє абсолютне відхилення **Mean absolute error = 0.68** – це абсолютне значення різниці між прогнозами, який дає **модель** для тестового набору і відомим для тестового набору.

Середнє абсолютне відхилення **Train mean absolute error = 0.64** - це абсолютне значення різниці між прогнозами, який дає **модель** для тренувального набору і **відомим** для тренувального набору.

Легко бачити, ці метрики якості навчання відповідні між тренувальним і тестовим наборами. Компроміс між складністю і узагальненістю досягнутий (див. вище).

Можна спробувати ускладнити модель, щоб покращити результат. Але відомо, що «**краще**» ворог якості.

## **ЗАВДАННЯ:**

**A)** Поновити код, наведений, як приклад.

Файл `data_singlevar_regr.txt` (див. Додаток) містить тренувальні дані у форматі ‘X, Y’

1. Отримати робочий код і результати.
2. Знайти інформаційні джерела і описати зміст метрик якості.

**B)** Самостійна робота

Файли `data_1.txt` або `data_2.txt` (див. Додаток) містять дані каротажних досліджень свердловин у форматі ‘Depth, Temperature’

1. Завантажити свій варіант даних (парний або непарний номер у списку групи). Візуалізувати фрагмент даних.
2. Утворити тренувальний та тестовий набори даних.
3. Побудувати модель лінійної регресії між глибиною точки виміру і температурою середовища у цій точці.
4. Візуалізувати дані, виділивши різними кольорами тренувальний, тестовий набори даних та лінію регресії.

5. Отримати метрики якості моделі й оцінити її узагальненість.

## Список літератури

1. Flach P. Machine Learning: The Art and Science of Algorithms that Make Sense of Data. - Cambridge University Press, 2012. – 396 p.  
<https://www.pdfdrive.com/machine-learnin00g-the-art-and-science-of-algorithms-that-make-sense-of-data-e178498797.html>Harrison M.
2. Illustrated Guide to Python 3: A Complete Walkthrough of Beginning Python with Unique Illustrations Showing how Python Really Works.  
<https://www.amazon.com/Illustrated-Guide-Python-Walkthrough-Illustrations/dp/1977921752>

## ДОДАТОК

При переході до безпосереднього виконання лабораторних робіт необхідно отримати у викладача доступ до файлів з даними

([e-Disk -> MashLearn -> MashLearn -> Master -> Master2021](#)).

Для цього:

- необхідно отримати поштову скриньку у е-мейлорі [nik\\_@ukr.net](mailto:nik_@ukr.net);
- на адресу надіслати свої дані (ПІБ, номер залікової книжки) на адресу [lal0220@ukr.net](mailto:lal0220@ukr.net) ;
- отримати посилання на файли з даними.