МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторної роботи
«Основи роботи з бібліотекою Pandas»

з курсу «Обробка даних Python»
для студентів спеціальностей 121 Інженерія програмного забезпечення,
122 Комп'ютерні науки, 124 Системний аналіз

Затверджено
редакційно-видавничою
радою університету,
протокол № 3 від 26.10.2022 року

Харків
НТУ «ХПІ»
2022

Методичні вказівки до лабораторної роботи «Основи роботи з бібліотекою Pandas» з курсу «Обробка даних Python» для студентів спеціальностей 121 Інженерія програмного забезпечення, 122 Комп'ютерні науки, 124 Системний аналіз / уклад. : С. М. Коваленко, С. В. Коваленко. – Харків : НТУ «ХПІ», 2022. – 44 с.

Укладачі: С. М. Коваленко,
С. В. Коваленко

Рецензент: І. П. Гамаюн

Кафедри програмної інженерії та інтелектуальних технологій управління, системного аналізу та інформаційно-аналітичних технологій

ВСТУП

Раніше ми обговорювали деталі NumPy та його об'єкта ndarray, який забезпечує ефективне зберігання та маніпулювання масивами в Python. Тут ми будемо спиратися на ці знання, детально розглядаючи структури даних, що надані бібліотекою Pandas. Pandas – пакет, побудований поверх NumPy, він забезпечує ефективну реалізацію DataFrame. DataFrame – це, по суті, багатовимірні масиви з прикріпленими мітками рядків і стовпців, і часто з неоднорідними типами та/або відсутніми даними.

Pandas – це швидкий, потужний, гнучкий і простий у використанні інструмент аналізу та обробки даних з відкритим кодом, побудований на основі мови програмування Python [1].

Pandas є корисним інструментом при роботі з табличними даними, що зберігаються в електронних таблицях або базах даних. Ця бібліотека допомагає досліджувати, очищати та обробляти дані. В Pandas таблиці даних називаються DataFrame. Pandas підтримує інтеграцію з багатьма форматами даних, такими як csv, excel, sql, json тощо.

Окрім зручного інтерфейсу зберігання маркованих даних, Pandas реалізує ряд потужних операцій з даними, знайомих користувачам як фреймворки баз даних, так і програм електронних таблиць. Як ми бачили, структура даних ndarray NumPy забезпечує важливі особливості для типу чистих, добре впорядкованих даних, що зазвичай спостерігаються в числових обчислювальних завданнях. Незважаючи на те, що ndarray слугує цій меті дуже добре, його обмеження стають зрозумілими, коли нам потрібна більша гнучкість (наприклад, прикріплення міток до даних, робота з відсутніми даними тощо) та при спробах операцій, які погано відображаються на поелементному бродкастінгу (наприклад, групування, зведені таблиці тощо), кожна з яких є важливою частиною аналізу менш структурованих даних, доступних у багатьох формах у навколишньому світі. Pandas, зокрема його об'єкти Series і DataFrame, базується на структурі масиву NumPy і забезпечує ефективний доступ до таких видів завдань «зміни даних», які займають більшу частину часу дата аналітика. У цій частині курсу ми зупинимось на механіці ефективного використання

`Series`, `DataFrame` та суміжних структур. Ми будемо використовувати приклади, отримані з реальних наборів даних, де це є доречним.

Мета: отримати базові знання та навички з аналізу та обробки даних за допомогою бібліотеки Pandas.

1. ТЕОРЕТИЧНІ ОСНОВИ

1.1 Встановлення та використання бібліотеки Pandas

Для встановлення Pandas у вашій системі потрібно встановити NumPy. Детально про встановлення можна ознайомитись у Pandas documentation [2]. Якщо ви слідували порадам, викладеним у попередніх лекціях, і використовували стек Anaconda, ви вже встановили Pandas. Після встановлення Pandas ви можете імпортувати його та перевірити версію [3]:

```
In [1]: import pandas
        pandas.__version__
Out[1]: '1.0.1'
```

Подібно до того, як ми зазвичай імпортуємо NumPy під псевдонімом `np`, ми будемо імпортувати Pandas під псевдонімом `pd`:

```
In [2]: import pandas as pd
```

1.2 Введення до об’єктів Pandas

На самому базовому рівні об’єкти Pandas можна розглядати як вдосконалені версії структурованих масивів NumPy [3], в яких рядки та стовпці ідентифікуються мітками, а не простими цілочисельними індексами. Як ми побачимо, Pandas пропонує велику кількість корисних інструментів, методів та функціональних можливостей поверх базових структур даних, але майже все, що буде використаним, вимагатиме розуміння того, що це за структури. Отже, перед тим, як йти далі, давайте представимо ці три основні структури даних Pandas:

`Series` , `DataFrame` та `Index`.

Ми розпочнемо наші зі стандартного імпортування NumPy та Pandas:

```
In [3]: import numpy as np
        import pandas as pd
```

1.3 Об'єкт `Series` Pandas

Pandas `Series` – це одновимірний масив індексованих даних. Його можна створити зі списку або масиву наступним чином:

```
In [4]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
        data
Out[4]: 0 0.25
         1 0.50
         2 0.75
         3 1.00
         dtype: float64
```

Як ми бачимо в `Output`, `Series` являє собою і послідовність значень, і послідовність індексів, до яких ми можемо отримати доступ за значеннями та атрибутами індексу. Ці значення – це просто звичний масив NumPy:

```
In [5]: data.values
Out[5]: array([0.25, 0.5 , 0.75, 1. ])
```

`index` – це масивноподібний об'єкт типу `pd.Index`, про який ми поговоримо докладніше

```
In [6]: data.index
Out[6]: RangeIndex(start=0, stop=4, step=1)
```

Як і в масиві NumPy, доступ до даних можна отримати за допомогою відповідного індексу через звичну нотацію квадратних

дужок Python:

```
In [7]: data[1]
Out[7]: 0.5
In [8]: data[1:3]
Out[8]: 1 0.50
        2 0.75
        dtype: float64
```

Однак, як ми побачимо, об'єкт `Series` Pandas набагато більш загальний та гнучкий, ніж одновимірний масив NumPy, який він емулює.

1.3.1 Series як узагальнений масив NumPy

З того, що ми тільки що побачили, може виглядати, що об'єкт `Series` в основному взаємозамінний з одновимірним масивом NumPy. Суттєвою відмінністю є наявність індексу: в той час як масив Numpy має *неявно визначений* цілочисельний індекс, що використовується для доступу до значень, Pandas `Series` має *явно визначений* індекс, пов'язаний зі значеннями.

Це явне визначення індексу надає об'єкту `Series` додаткові можливості. Наприклад, індекс не обов'язково повинен бути цілим числом, але він може складатися зі значень будь-якого бажаного типу. Наприклад, якщо ми хочемо, ми можемо використовувати рядки як індекс:

```
In [9]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                         index=['a', 'b', 'c', 'd'])
        data
Out[9]: a 0.25
        b 0.50
        c 0.75
        d 1.00
         dtype: float64
```

```
In [10]: data.index
Out[10]: Index(['a', 'b', 'c', 'd'],
          dtype='object')
```

І доступ до елементів працює як і раніше:

```
In [11]: data['b']
Out[11]: 0.5
```

Ми можемо використовувати навіть несуміжні або непослідовні індекси:

```
In [12]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                              index=[2, 5, 3, 7])
         data
Out[12]: 2 0.25
         5 0.50
         3 0.75
         7 1.00
         dtype: float64
```

1.3.2 Series як спеціалізований словник

Таким чином, ви можете уявити собі `Series` Pandas, трохи схожим на особливу реалізацію словника Python. Словник – це структура, яка відображає довільні ключі до набору довільних значень, а `Series` – це структура, яка ставить у відність типизовані ключі до набору типизованих значень. Ця типизація важлива: так само, як скомпільований за типом код масиву NumPy робить його ефективнішим, ніж список Python для певних операцій, інформація про тип Pandas `Series` робить його набагато ефективнішим, ніж словники Python для певних операцій.

Аналогію `Series` як словник можна зробити ще більш зрозумілою, побудувавши об’єкт `Series` безпосередньо зі словника Python:

```
In [13]: population_dict = {'Kharkiv': 1419036,
```

```
                              'Kherson': 289697,
                              'Mariupol': 446103,
                              'Lviv': 721301,
                              'Kyiv': 2884359}
         population = pd.Series(population_dict)
         population
Out[13]: Kharkiv     1419036
         Kherson      289697
         Mariupol     446103
         Lviv         721301
         Kyiv        2884359
         dtype: int64
```

За замовчуванням буде створено `Series`, де індекс виводиться з ключів. Звідси можна здійснити типовий доступ до елементів у стилі словника:

```
In [14]: population['Kharkiv']
Out[14]: 1419036
```

Однак, на відміну від словника, `Series` також підтримує операції в стилі масиву, такі як слайсинг:

```
In [14]: population['Kherson':'Lviv']
Out[14]: Kherson     289697
         Mariupol    446103
         Lviv        721301
         dtype: int64
```

1.3.3 Побудова об’єктів Series

Ми вже бачили кілька способів побудови Pandas `Series` з нуля; всі вони є деякою версією наступного:

```
In[]: pd.Series(data, index=index),
```

де `index` є необов’язковим аргументом, а `data` можуть бути однією з

багатьох сутностей.

Наприклад, `data` можуть бути списком або масивом NumPy, у цьому випадку за замовчуванням `index` має цілочисельну послідовність:

```
In [15]: pd.Series([2, 4, 6])
Out[15]: 0    2
         1    4
         2    6
         dtype: int64
```

`data` може бути скаляром, який повторюється для заповнення вказаного індексу:

```
In [16]: pd.Series(5, index=[100, 200, 300])
Out[16]: 100    5
         200    5
         300    5
         dtype: int64
```

`data` може бути словником, в якому `index` за замовчуванням використовує ключі словника:

```
In [17]: pd.Series({2:'a', 1:'b', 3:'a'})
Out[17]: 2    a
         1    b
         3    a
         dtype: object
```

У кожному випадку індекс може бути явно встановлений, якщо бажано отримати інший результат:

```
In [18]:pd.Series({2:'a',1:'b',3:'c'}, index=[3,2])
Out[18]: 3    c
         2    a
         dtype: object
```

Зверніть увагу, що в цьому випадку `Series` заповнюється лише чітко визначеними ключами.

1.4 Об'єкт `DataFrame`

Якщо `Series` є аналогом одновимірного масиву з гнучкими індексами, `DataFrame` – це аналог двовимірного масиву як з гнучкими індексами рядків, так і з гнучкими іменами стовпців.

1.4.1 DataFrame як узагальнений масив NumPy

Подібно до того, як ви можете розглядати двовимірний масив як упорядковану послідовність одновимірних стовпців, можна представити `DataFrame` як послідовність об'єктів `Series`.

Щоб продемонструвати це, давайте спочатку побудуємо новий `Series` із переліком площі кожного з п'яти міст України, про які йшлося в попередньому розділі:

```
In [19]: area_dict = {'Kharkiv': 350,
                      'Kherson': 145,
                      'Mariupol': 135,
                      'Lviv': 149,
                      'Kyiv': 856}
         area = pd.Series(area_dict)
         area
Out[19]: Kharkiv     350
         Kherson     145
         Mariupol    135
         Lviv        149
         Kyiv        856
         dtype: int64
```

Тепер, коли ми маємо це разом із `Series population`, ми можемо використати словник для побудови єдиного двовимірного об'єкта, що містить інформацію щодо населення та площі міст України:

```
In [20]: cities=pd.DataFrame({'population':population,
```

```
                              'area': area})
        cities
Out[20]:
```

	population	area
Kharkiv	1419036	350
Kherson	289697	145
Mariupol	446103	135
Lviv	721301	149
Kyiv	2884359	856

Як і об'єкт `Series`, `DataFrame` має атрибут `index`, що надає доступ до міток індексу:

```
In [21]: cities.index
Out[21]: Index(['Kharkiv', 'Kherson', 'Mariupol',
         'Lviv', 'Kyiv'], dtype='object')
```

Крім того, `DataFrame` має атрибут `columns`, який є об'єктом `Index`, що містить мітки стовпців:

```
In [22]: cities.columns
Out[22]: Index(['population', 'area'], dtype='object')
```

Таким чином, `DataFrame` можна розглядати як узагальнення двовимірного масиву NumPy, де і рядки, і стовпці мають узагальнений індекс для доступу до даних.

1.4.2 DataFrame як спеціалізований словник

Подібним чином ми можемо розглядати `DataFrame` як спеціалізацію словника. Там, де словник ставить у відповідність ключ до значення, `DataFrame` ставить у відповідність ім'я стовпця до `Series` даних стовпців. Наприклад, запит атрибуту `area` повертає об'єкт `Series`, що містить площі, які ми бачили раніше:

```
In [23]: cities['area']
Out[23]: Kharkiv     350
         Kherson     145
         Mariupol    135
         Lviv        149
         Kyiv        856
         Name: area, dtype: int64
```

Зверніть увагу на потенційну можливість плутанини: у двовимірному масиві NumPy `data[0]` поверне перший рядок. Для `DataFrame`, `data['col0']` поверне перший стовпець. Через це, мабуть, краще думати про `DataFrame` як про узагальнені словники, а не як про узагальнені масиви, хоча обидва способи розгляду ситуації можуть бути корисними

1.5 Побудова об'єктів `DataFrame`

Pandas `DataFrame` може бути побудовано різними способами. Тут ми наведемо кілька прикладів.

1.5.1 З одного об'єкта Series

`DataFrame` – це колекція `Series` об'єктів, і `DataFrame` з одним стовпчиком може бути побудований з одного `Series`:

```
In[24]:pd.DataFrame(population,columns=['population'])
Out[24]:
```

	population
Kharkiv	1419036
Kherson	289697
Mariupol	446103
Lviv	721301
Kyiv	2884359

1.5.2 Зі списку словників

З будь-якого списку словників можна створити `DataFrame`. Ми використаємо просте заповнення списку для створення даних:

```
In [25]: data = [{'a': i, 'b': 2 * i}
                 for i in range(5)]
         pd.DataFrame(data)
Out[25]:
```

	a	b
0	0	0
1	1	2
2	2	4
3	3	6
4	4	8

Навіть якщо в словнику відсутні деякі ключі, Pandas заповнить їх значеннями `NaN` (тобто "`not a number`"):

```
In [26]: pd.DataFrame([{'a': 1, 'b': 2},
                       {'b': 3, 'c': 4}])
Out[26]:
```

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

1.5.3 Зі словника об’єктів Series

Як ми бачили раніше, `DataFrame` також може бути побудований зі словника об’єктів `Series`:

```
In [27]: pd.DataFrame({'population': population,
                       'area': area})
Out[27]:
```

	population	area
Kharkiv	1419036	350
Kherson	289697	145
Mariupol	446103	135
Lviv	721301	149
Kyiv	2884359	856

1.5.4 З двовимірного масиву NumPy

Маючи двовимірний масив даних, ми можемо створити `DataFrame` з будь-якими вказаними іменами стовпців та індексів. Якщо їх не вказувати, для кожного із стовпців буде використовуватися цілочисельний індекс:

```
In [28]: pd.DataFrame(np.random.rand(3, 2),
                      columns=['foo', 'bar'],
                      index=['a', 'b', 'c'])
Out[28]:
```

	foo	bar
a	0.780121	0.846270
b	0.166138	0.396992
c	0.846922	0.827634

1.6 Об’єкт Pandas `Index`

Ми бачили, що об’єкти `Series` і `DataFrame` містять явний індекс, що дозволяє посилатися та змінювати дані. Цей об’єкт `Index` сам по собі є цікавою структурою, і його можна сприймати і як незмінний масив, так і як упорядковану множину (технічно мультімножину, оскільки об’єкти `Index` можуть містити повторювані значення). Це має деякі цікаві наслідки в операціях, доступних над об’єктами `Index`. Як простий приклад, давайте побудуємо `Index` зі списку цілих чисел:

```
In [29]: ind = pd.Index([2, 3, 5, 5, 7, 11])
         ind
Out[29]: Int64Index([2, 3, 5, 5, 7, 11],dtype='int64')
In [30]: df_ex=pd.DataFrame(['a','b','c','d','e','f'],
                                 index=ind)
         df_ex
Out[30]:
```

	0
2	a
3	b
5	c
5	d
7	e
11	f

1.6.1 `Index` як незмінний масив

`Index` багато в чому працює як масив. Наприклад, ми можемо використовувати стандартну нотацію індексації Python для отримання значень або фрагментів:

```
In [31]: ind[1]
Out[31]: 3
In [32]: ind[::2]
Out[32]: Int64Index([2, 5, 7], dtype='int64')
```

Index об'єкти також мають багато атрибутів, знайомих з масивів NumPy:

```
In [33]: print(ind.size, ind.shape, ind.ndim,
                   ind.dtype)
Out[33]: 6 (6,) 1 int64
```

Однією з різниць між об'єктами `Index` та масивами NumPy є те, що індекси незмінні – тобто їх не можна змінювати звичайними

засобами:

```
In [34]: try:
             ind[1] = 0
         except TypeError as e:
             print (f'TypeError: ',e)
Out[34]: TypeError: Index does not support mutable
operations
```

Ця незмінність робить більш безпечним обмін індексами між кількома `DataFrame` та масивами, без потенційних побічних ефектів від ненавмисної модифікації індексу.

1.6.2 `Index` як множина

Об’єкти Pandas призначені для полегшення таких операцій, як об'єднання наборів даних, які залежать від багатьох аспектів арифметики множин. Об’єкт `Index` дотримується багатьох домовленостей, використовуваних вбудованою структурою даних `set` Python, так що об’єднання, перетини, відмінності та інші комбінації можуть бути обчислені звичним способом:

```
In [35]: indA = pd.Index([1, 3, 5, 7, 9])
         indB = pd.Index([2, 3, 5, 7, 11])
In [36]: indA.intersection(indB)
Out[36]: Int64Index([3, 5, 7], dtype='int64')
In [37]: indA.union(indB)
Out[37]: Int64Index([1, 2, 3, 5, 7, 9, 11],
         dtype='int64')
In [38]: indA.symmetric_difference(indB)
Out[38]: Int64Index([1, 2, 9, 11], dtype='int64'))
```

1.7 Індексація та вибір даних

Раніше ми детально розглядали методи та засоби доступу, встановлення та модифікації значень у масивах NumPy. Сюди входили індексація (наприклад, `arr[2, 1]`), слайсінг (наприклад, `arr[:, 1:5]`), маски (наприклад, `arr[arr > 0]`), розширене

індексування (fancy indexing) (наприклад, `arr[0, [1, 5]]`) та їх комбінації (наприклад,`arr[:, [1, 5]]`). Зараз ми розглянемо подібні засоби доступу та модифікації значень у об'єктах Pandas `Series` і `DataFrame`. Якщо ви використовували шаблони NumPy, відповідні шаблони в Pandas будуть відчуватися дуже знайомими, хоча є кілька відмінностей, про які слід пам'ятати.

Почнемо з простого випадку одновимірного об'єкта `Series`, а потім перейдемо до більш складного двовимірного об'єкта `DataFrame`.

1.7.1 Вибір даних у `Series`

Як ми бачили в попередньому розділі, об'єкт `Series` багато в чому діє як одновимірний масив NumPy і багато в чому як стандартний словник Python. Будемо пам'ятати про ці дві аналогії, що перекриваються, і це допоможе нам зрозуміти закономірності індексації та відбору даних у цих масивах.

Як і словник, об'єкт `Series` забезпечує співставлення набору ключів до набору значень:

```
In [39]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                          index=['a', 'b', 'c', 'd'])
         data
Out[39]: a    0.25
         b    0.50
         c    0.75
         d    1.00
         dtype: float64
In [40]: data['b']
Out[40]:0.5
```

Ми також можемо використовувати словникові вирази та методи Python для дослідження ключів/індексів та значень:

```
In [41]: 'a' in data
Out[41]: True
In [42]: data.keys()
Out[42]: Index(['a', 'b', 'c', 'd'],dtype='object')
```

```
In [43]: list(data.items())
Out[43]: [('a', 0.25), ('b', 0.5), ('c', 0.75),
         ('d', 1.0)]
```

`Series` об'єкти можна навіть модифікувати за допомогою словникового синтаксису. Подібно до того, як ви можете розширити словник, призначивши новий ключ, ви можете продовжити `Series`, призначивши нове значення індексу:

```
In [44]: data['e'] = 1.25
         data
Out[44]: a    0.25
         b    0.50
         c    0.75
         d    1.00
         e    1.25
         dtype: float64
```

Ця легка зміна об'єктів є зручною особливістю: під капотом Pandas приймає рішення щодо розміщення пам'яті та копіювання даних, які можуть знадобитися; користувачеві, як правило, не потрібно турбуватися про ці проблеми.

`Series` базується на словниковому інтерфейсі та забезпечує вибір елементів у стилі масиву за допомогою тих самих основних механізмів, що й масиви NumPy – тобто зрізи, маски та fancy індексування. Приклади:

```
In [45]: # Зрізи по явному індексу
        data['a':'c']
Out[45]:a    0.25
        b    0.50
        c    0.75
        dtype: float64
In [46]: # Зрізи по неявному індексу
        data[0:2]
Out[46]:a    0.25
```

```
        b    0.50
        dtype: float64
In [47]: # Використання маски
        data[(data > 0.3) & (data < 0.8)]
Out[47]: b    0.50
        c    0.75
        dtype: float64

In [48]: # Розширене індексування
        data[['a', 'e']]
Out[48]: a    0.25
        e    1.25
        dtype: float64
```

Серед всіх цих інструментів, зрізи можуть спричинити найбільшу плутанину. Зверніть увагу, що при слайсингу *з явним індексом* (тобто `data ['a': 'c']`), кінцевий індекс *включається* в зріз, тоді як при зрізах з *неявним індексом* (тобто `data[0:2]`), остаточний індекс *виключається* зі зрізу.

1.7.2 Індексатори: `loc` *і* `iloc`

Правила слайсингу та індексування можуть викликати плутанину. Наприклад, якщо у вашому `Series` є явний цілочисельний індекс, операція індексації, така як `data[1],` використовуватиме явні індекси, тоді як операція слайсингу, як `data[1: 3]` буде використовувати неявний індекс у стилі Python.

```
In [49]: data = pd.Series(['a', 'b', 'c'],
                          index=[1, 3, 5])
         data
Out[49]: 1    a
         3    b
         5    c
         dtype: object
In [50]: # Використання явного індексу для доступу
         data[1]
```

```
Out[50]: 'a'
In [51]: # Використання неявного індексу для зрізів
         data[1:3]
Out[51]: 3    b
         5    c
         dtype: object
```

Через цю потенційну плутанину у випадку цілочисельних індексів, Pandas надає деякі спеціальні атрибути `indexer`, які явно розкривають певні схеми індексації. Це не функціональні методи, а атрибути, які надають певний інтерфейс нарізки даним у `Series`.

По-перше, атрибут `loc` дозволяє індексувати та нарізати, з посиланням на *явний* індекс:

```
In [52]: data.loc[1]]
Out[52]: 'a'
In [53]: data.loc[1:3]
Out[53]: 1    a
         3    b
         dtype: object
```

`iloc` атрибут дозволяє індексувати та нарізати, з посиланням на неявний індекс у стилі Python:

```
In [52]: data.iloc[1]]
Out[52]: 'b'
In [53]: data.iloc[1:3]
Out[53]: 3    b
         5    c
         dtype: object
```

Одним із керівних принципів Python, є «явне краще, ніж неявне» (“Explicit is better than implicit”). Явна природа `loc` та `iloc` робить їх дуже корисними для підтримки чистого та читабельного коду, особливо у випадку цілочисельних індексів. Тому використання цих атрибутів є рекомендованим як для полегшення читання та розуміння

коду, так і для запобігання дрібним помилкам.

1.7.3 Вибір даних у `DataFrame`

Нагадаємо, що `DataFrame` з одного боку діє як двовимірний або структурований масив, з іншого – як словник структур `Series`, що мають один і той же індекс. Ці аналогії можуть бути корисними, коли ми досліджуємо відбір даних у цій структурі.

Першою аналогією, яку ми розглянемо, є `DataFrame` як словник пов'язаних об’єктів `Series`. Повернемося до нашого прикладу площі і населення міст України:

```
In [54]: area = pd.Series({'Kharkiv': 350,
                          'Kherson': 145,
                                 'Mariupol': 135,
                          'Lviv': 149,
                          'Kyiv': 856})
         pop = pd.Series({'Kharkiv': 1419036,
                          'Kherson': 289697,
                          'Mariupol': 446103,
                          'Lviv': 721301,
                          'Kyiv': 2884359})
  data=pd.DataFrame({'area':area,'pop':pop})
  data
Out[54]:
```

	area	pop
Kharkiv	350	1419036
Kherson	145	289697
Mariupol	135	446103
Lviv	149	721301
Kyiv	856	2884359

Окремі `Series`, що складають стовпці `DataFrame`, можна отримати за допомогою індексації назви стовпця у стилі словника:

```
In [55]: data['area']
```

```
Out[55]: Kharkiv     350
         Kherson     145
         Mariupol    135
         Lviv        149
         Kyiv        856
         Name: area, dtype: int64
```

Крім того, ми можемо використовувати доступ до стовпчиків як до атрибутів `DataFrame`

```
In [56]: data.area
Out[56]: Kharkiv     350
         Kherson     145
         Mariupol    135
         Lviv        149
         Kyiv        856
         Name: area, dtype: int64
```

Цей доступ до стовпця як до атрибутів насправді отримує такий самий об'єкт, як і доступ у стилі словника:

```
In [57]: data.area is data['area']
Out[57]: True
```

Хоча це корисне скорочення, майте на увазі, що воно працює не у всіх випадках! Наприклад, якщо імена стовпців не є рядками, або якщо імена стовпців конфліктують з методами `DataFrame`, такий тип доступу неможливий. Наприклад, у `DataFrame` є метод `pop()`, тому `data.pop` вказуватиме на метод, а не на стовпець `pop`:

```
In [58]: data.pop is data['pop']
Out[58]: False
```

Як і для обговорених раніше об'єктів `Series`, синтаксис у стилі словника також може бути використаний для модифікації об'єкта, в цьому випадку додаючи новий стовпець:

```
In [59]: data['density'] = data['pop']/data['area']
data
Out[59]:
```

	area	pop	density
Kharkiv	350	1419036	4054.388571
Kherson	145	289697	1997.910345
Mariupol	135	446103	3304.466667
Lviv	149	721301	4840.946309
Kyiv	856	2884359	3369.578271

Як вже згадувалося раніше, ми також можемо розглядати `DataFrame` як вдосконалений двовимірний масив. Ми можемо перевірити базовий масив даних, використовуючи атрибут `values`:

```
In [60]:data.values[3,2] #щильність населення у Львові
Out[60]: 4840.946308724832
```

Маючи це на увазі, багато відомих дій, за аналогією з масивом, можна зробити на самому `DataFrame`. Наприклад, ми можемо транспонувати повний `DataFrame`, щоб поміняти місцями рядки та стовпці:

```
In [61]: data.T
Out[61]:
```

	Kharkiv	Kherson	Mariupol	Lviv	Kyiv
area	3.500000e+02	145.000000	135.000000	149.000000	8.560000e+02
pop	1.419036e+06	289697.000000	446103.000000	721301.000000	2.884359e+06
density	4.054389e+03	1997.910345	3304.466667	4840.946309	3.369578e+03

Що стосується індексації об’єктів `DataFrame`, то очевидно, що індексація стовпців у стилі словника виключає можливість просто розглядати це як масив NumPy. Зокрема, передача одного індексу масиву отримує доступ до рядка:

```
In [62]: data.values[0]
Out[62]: array([3.50000000e+02, 1.41903600e+06,
                4.05438857e+03])
```

Передача одного `index` в `DataFrame` дає доступ до стовпця:

```
In [63]: data['area']
Out[63]: Kharkiv     350
         Kherson     145
         Mariupol    135
         Lviv        149
         Kyiv        856
         Name: area, dtype: int64
```

Таким чином, для індексації в стилі масиву нам потрібна інша умова. Тут Pandas знову використовує згадані раніше індексатори `loc`, `iloc`. Використовуючи індексатор `iloc`, ми можемо індексувати базовий масив так, ніби це простий масив NumPy (з використанням неявного індексу), але в результаті зберігаються мітки індексів рядка та стовпця `DataFrame`:

```
In [64]: data.iloc[:3, :2]
Out[64]:
```

	area	pop
Kharkiv	350	1419036
Kherson	145	289697
Mariupol	135	446103

Подібним чином, використовуючи індексатор `loc`, ми можемо індексувати дані у стилі, подібному до масиву, але використовуючи явні імена індексів та стовпців:

```
In [65]: data.loc[:'Mariupol', :'pop']
Out[65]:
```

	area	pop
Kharkiv	350	1419036
Kherson	145	289697
Mariupol	135	446103

Будь-який із звичних шаблонів доступу до даних у стилі NumPy можна використовувати в цих індексаторах. Наприклад, в індексаторі `loc` ми можемо поєднувати використання масок та розширеного індексування, як показано нижче:

```
In [66]: data.loc[data.density>3000,['pop','density']]
Out[66]:
```

	pop	density
Kharkiv	1419036	4054.388571
Lviv	721301	4840.946309
Kyiv	2884359	3369.578271

Будь-яка з цих домовленостей щодо індексування також може використовуватися для встановлення або модифікації значень; це робиться стандартним способом, до якого ви могли б звикнути працювати з NumPy:

```
In [67]: data.iloc[2, 2] = 900
         data
Out[67]:
```

	area	pop	density
Kharkiv	350	1419036	4054.388571
Kherson	145	289697	1997.910345
Mariupol	135	446103	900.000000
Lviv	149	721301	4840.946309
Kyiv	856	2884359	3369.578271

Існує кілька додаткових правил щодо індексування, які можуть

здатися розбіжними з попереднім обговоренням, але тим не менше можуть бути дуже корисними на практиці.

По-перше, тоді як *індексація* відноситься до стовпців, *слайсинг* відноситься до рядків:

```
In [68]: data['Kharkiv':'Mariupol']
Out[68]:
```

	area	pop	density
Kharkiv	350	1419036	4054.388571
Kherson	145	289697	1997.910345
Mariupol	135	446103	900.000000

Такі фрагменти також можуть посилатися на рядки за номером, а не за індексом:

```
In [69]: data[1:3]
Out[69]:
```

	area	pop	density
Kherson	145	289697	1997.910345
Mariupol	135	446103	900.000000

Подібним чином, операції застосування маски також інтерпретуються по рядках, а не по стовпцях:

```
In [70]: data[data.density > 4000]
Out[70]:
```

	area	pop	density
Kharkiv	350	1419036	4054.388571
Lviv	149	721301	4840.946309

Якщо ж необхідно отримати доступ до кількох стовпчиків, то можна скористатися наступними методами:

```
In [71]: data[['area','pop']]
```

```
Out[71]:
```

	area	pop
Kharkiv	350	1419036
Kherson	145	289697
Mariupol	135	446103
Lviv	149	721301
Kyiv	856	2884359

```
In [72]: data.loc[:, 'area':'pop']
Out[72]:
```

	area	pop
Kharkiv	350	1419036
Kherson	145	289697
Mariupol	135	446103
Lviv	149	721301
Kyiv	856	2884359

1.8 Операція з даними в Pandas

Однією з найважливіших частин NumPy є здатність виконувати швидкі елементарні операції, як з базовою арифметикою (додавання, віднімання, множення тощо), так і з більш складними операціями (тригонометричні функції, експоненційні та логарифмічні функції тощо). Pandas успадковує більшу частину цієї функціональності від NumPy.

Pandas включає в себе кілька корисних функцій: для унарних операцій, таких як заперечення та тригонометричні функції, ці універсальні функції будуть зберігати мітки індексів рядків та стовпців на виході, а для бінарних операцій, таких як додавання та множення, Pandas автоматично «вирівнює» індекси, при передачі об’єктів до універсальних функцій. Це означає, що збереження вмісту даних та комбінування даних з різних джерел – що є потенційно схильніми до помилок при роботі з необробленими масивами NumPy – стають по суті надійними із Pandas. Додатково ми побачимо, що існують чітко визначені операції між одновимірними структурами `Series` та

двовимірними структурами `DataFrame`.

1.8.1 Універсальні функції: збереження індексу

Оскільки Pandas призначений для роботи з NumPy, будь-які універсальні функції NumPy будуть працювати у Pandas `Series` та `DataFrame` об'єктах. Почнемо з визначення простих `Series` та `DataFrame`, на яких це можна продемонструвати:

```
In [73]: rnd = np.random.RandomState(10)
         a = pd.Series(rnd.randint(0, 10, 4))
         a
Out[73]: 0    9
         1    4
         2    0
         3    1
         dtype: int32
```

Якщо ми застосуємо універсальні функції NumPy цього об'єкта, результатом буде інший об'єкт Pandas із збереженими індексами:

```
In [74]: np.exp(a)
Out[74]: 0    8103.083928
         1      54.598150
         2       1.000000
         3       2.718282
         dtype: float64
```

Аналогічний результат отримаємо і з об'єктом `DataFrame:`

```
In [75]: df = pd.DataFrame(rnd.randint(0, 10, (3, 4)),
                           columns=['a','b','c','d'])
        df
Out[75]:
```

	a	b	c	d
0	6	8	1	8
1	4	1	3	6
2	5	3	9	6

```
In [76]: np.square(df)
Out[76]:
```

	a	b	c	d
0	36	64	1	64
1	16	1	9	36
2	25	9	81	36

1.8.1 Універсальні функції: Вирівнювання індексу (Index Alignment)

Для бінарних операцій над двома об'єктами `Series` або `DataFrame`, Pandas вирівнює індекси в процесі виконання операції. Це дуже зручно під час роботи з неповними даними.

Почнемо розгляд з об'єктів `Series`.

Як приклад, припустимо, ми поєднуємо два різні джерела даних і знаходимо лише чотири міста України за площею і чотири міста за населенням:

```
In [77]: area = pd.Series({'Kharkiv': 350,
                           'Mariupol': 135,
                           'Lviv': 149,
                           'Kyiv': 856})
         population= pd.Series({'Kharkiv': 1419036,
                                'Kherson': 289697,
                                'Mariupol': 446103,
                                'Kyiv': 2884359})
```

Давайте подивимося, що відбувається, коли ми розділимо ці значення для обчислення щільності населення:

```
In [78]: population / area
```

```
Out[78]: Kharkiv     4054.388571
         Kherson             NaN
         Kyiv        3369.578271
         Lviv                NaN
         Mariupol    3304.466667
         dtype: float64
```

Отриманий масив містить об'єднання індексів двох вхідних масивів, які можна визначити, використовуючи стандартну арифметику для множин Python для цих індексів:

```
In [79]: area.index.union(population.index)
Out[79]: Index(['Kharkiv', 'Kherson', 'Kyiv',
        'Lviv', 'Mariupol'], dtype='object')
```

Будь-який елемент, для якого той чи інший не має запису, позначається `NaN`, або `Not a Number` (не число), саме так Pandas позначає відсутні дані. Цей збіг індексів реалізовано таким чином для будь-якого вбудованого арифметичного виразу Python; будь-які відсутні значення за замовчуванням заповнюються `NaN`:

```
In [80]: A = pd.Series([1, 3, 5], index=[1, 2, 3])
         B = pd.Series([2, 4, 6], index=[0, 1, 2])
         A + B
Out[80]: 0    NaN
         1    5.0
         2    9.0
         3    NaN
         dtype: float64
```

Якщо використання значень `NaN` не є бажаним, можна задати значення для заповнення, використовуючи відповідні об’єктні методи замість операторів. Наприклад, виклик `A.add(B)`еквівалентно виклику `A + B`, але дозволяє необов'язкову явну вказівку значення заливки для будь-яких елементів в `A` або `B` які можуть бути відсутніми:

```
In [81]: A.add(B, fill_value=0)
Out[81]: 0    2
         1    5.0
         2    9.0
         3    5
         dtype: float64
```

При виконанні операцій між `DataFrame` та `Series` аналогічно підтримується вирівнювання індексу та стовпця. Операції між `DataFrame` та `Series` схожі на операції між двовимірним та одновимірним масивами NumPy. Розглянемо одну типову операцію, коли ми знаходимо різницю двовимірного масиву та одного з його рядків:

```
In [82]: A=pd.DataFrame(rnd.randint(0, 10, (2, 2)),
                              columns=list('AB'))
        A
Out[82]:
```

	A	B
0	9	1
1	9	4

```
In [83]: B=pd.DataFrame(rnd.randint(0, 10, (3, 3)),
                              columns=list('BAC'))
        B
Out[83]:
```

	B	A	C
0	7	9	3
1	5	2	4
2	7	6	8

```
In [84]: A + B
Out[84]:
```

	A	B	C
0	18.0	8.0	NaN
1	11.0	9.0	NaN
2	NaN	NaN	NaN

Зверніть увагу, що індекси правильно вирівняні незалежно від їх порядку в двох об'єктах, до того ж індекси в результаті сортуються. Як і у випадку із `Series`, ми можемо використовувати відповідні арифметичні методи та передати будь-яке бажане `fill_value`, яке буде використано замість відсутніх значень. Тут ми заповнимо середнім арифметичним всіх значень в `A` (обчислюється шляхом складання рядків `A`):

```
In [85]: fill = A.stack().mean()
         A.add(B, fill_value=fill)
Out[85]:
```

	A	B	C
0	18.00	8.00	8.75
1	11.00	9.00	9.75
2	11.75	12.75	13.75

2. ПОСТАНОВА ЗАДАЧІ

1. Завантажте набір даних про дитячі імена США з веб-сайту kaggle.com (https://www.kaggle.com/kaggle/us-baby-names?select=NationalNames.csv)

2. Виконайте вправи по варіантах. Для розрахунку номеру варіанту скористуйтеся формулою. $N = (n + 4) \% 5 + 1$, де N – номер варіанту, n – Ваш номер у списку групи.

Варіант	Номери вправ
1	1, 2, 3, 5, 10, 11, 12, 13, 14, 15, 16, 17, 18, 21, 22, 23, 24, 26
2	3, 4, 5, 8, 9, 11, 12, 13, 14, 16, 17, 18, 19, 20, 22, 23, 24, 27
3	1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 18, 19, 20, 21, 23, 25, 27
4	1, 3, 6, 7, 8, 12, 13, 14, 15, 16, 17, 19, 20, 22, 24, 25, 26, 27
5	2, 4, 6, 7, 9, 10, 15, 16, 17, 18, 20, 21, 22, 23, 24, 25, 26, 27

Вправи:

1. Виведіть перші 8 рядків набору даних.

Очікуваний результат:

Out[3]:

	Id	Name	Year	Gender	Count
0	1	Mary	1880	F	7065
1	2	Anna	1880	F	2604
2	3	Emma	1880	F	2003
3	4	Elizabeth	1880	F	1939
4	5	Minnie	1880	F	1746
5	6	Margaret	1880	F	1578
6	7	Ida	1880	F	1472
7	8	Alice	1880	F	1414

2. Вивести останні 8 рядків набору даних.

Очікуваний результат:

Out[4]:

	Id	Name	Year	Gender	Count
1825425	1825426	Zo	2014	M	5
1825426	1825427	Zyeir	2014	M	5
1825427	1825428	Zyel	2014	M	5
1825428	1825429	Zykeem	2014	M	5
1825429	1825430	Zymeer	2014	M	5
1825430	1825431	Zymiere	2014	M	5
1825431	1825432	Zyran	2014	M	5
1825432	1825433	Zyrin	2014	M	5

3. Отримайте імена стовпців набору даних

Очікуваний результат:

```
Out[4]: Index(['Id', 'Name', 'Year', 'Gender', 'Count'], dtype='object')
```

4. Отримайте загальну інформацію про дані у наборі даних.

Очікуваний результат:

Out[5]:

	Id	Year	Count
count	1.825433e+06	1.825433e+06	1.825433e+06
mean	9.127170e+05	1.972620e+03	1.846879e+02
std	5.269573e+05	3.352891e+01	1.566711e+03
min	1.000000e+00	1.880000e+03	5.000000e+00
25%	4.563590e+05	1.949000e+03	7.000000e+00
50%	9.127170e+05	1.982000e+03	1.200000e+01
75%	1.369075e+06	2.001000e+03	3.200000e+01
max	1.825433e+06	2.014000e+03	9.968000e+04

5. Знайдіть кількість унікальних імен у наборі даних

Очікуваний результат:

Out[33]:

93889

6. Обчисліть кількість унікальних жіночих та чоловічих імен у цілому наборі даних

Очікуваний результат:

Out[37]:

	Name
Gender	
F	64911
M	39199

7. Знайдіть 5 найпопулярніших чоловічих імен у 2010 році

Очікуваний результат:

```
Out[45]:
```

	Id	Name	Year	Gender	Count
1677392	1677393	Jacob	2010	M	22082
1677393	1677394	Ethan	2010	M	17985
1677394	1677395	Michael	2010	M	17308
1677395	1677396	Jayden	2010	M	17152
1677396	1677397	William	2010	M	17030

8. Знайдіть найпопулярніше ім’я за результатами одного року (ім’я, для якого `Count` максимальне)

Очікуваний результат:

```
The name is 'Linda' in 1947
```

9. Підрахуйте кількість записів, для яких `Count` - мінімальне у наборі.

Очікуваний результат:

```
Out[10]: 254615
```

10. Підрахуйте кількість унікальних імен у кожному році

Очікуваний результат:

```
Out[26]:
```

	Name
Year	
1880	1889
1881	1830
1882	2012
1883	1962
1884	2158

11. Знайдіть рік із найбільшою кількістю унікальних імен.

Очікуваний результат:

```
Out[32]:
```

Year	Name
2008	32488

12. Знайдіть найпопулярніше ім’я в році з найбільшою кількістю унікальних імен (тобто у 2008 році)

Очікуваний результат:

```
Out[24]:

'Jacob'
```

13. Знайдіть рік, коли ім’я “Jacob” було найпопулярнішим серед жіночих імен

Очікуваний результат:

	Id	Name	Year	Gender	Count
1455556	1455557	Jacob	2004	F	171

14. Знайти рік із найбільшою кількістю гендерно нейтральних імен (однакові чоловічі та жіночі імена)

Очікуваний результат:

```
Out[19]:
```

Year	Gender_neutral_names
2008	2557

15. Знайдіть загальну кількість народжень за рік.

Очікуваний результат для перших 5 рядків:

Out[56]:

	Count
Year	
1880	201484
1881	192699
1882	221538
1883	216950
1884	243467

16. Знайдіть рік, коли народилося найбільше дітей

Очікуваний результат:

Out[49]:

1957

17. Знайдіть кількість дівчаток та хлопчиків, які народились кожного року

Очікуваний результат для перших 5 рідків:

Out[50]:

Gender	F	M
Year		
1880	90993	110491
1881	91954	100745
1882	107850	113688
1883	112321	104629
1884	129022	114445

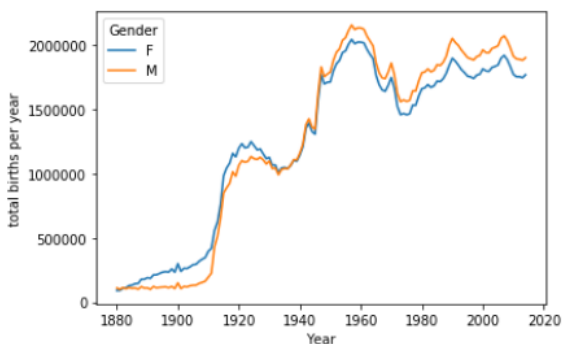
18. Підрахуйте кількість років, коли дівчаток народжувалось більше, ніж хлопчиків.

Очікуваний результат:

```
Out[64]: 54
```

19. Накресліть графік загальної кількості народжень хлопчиків та дівчаток на рік.

Очікуваний результат:



20. Підрахуйте кількість гендерно-нейтральних імен (однакових для дівчат та хлопців)

Очікуваний результат:

```
Out[85]: 10221
```

21. Порахуйте, скільки разів хлопчиків називали Barbara

Очікуваний результат:

```
Out[99]: 4139
```

22. Підрахуйте скільки років проводилось спостереження

Очікуваний результат:

```
Out[218]: 'Спостереження проводилось 135 років'
```

23. Знати найпопулярніші гендерно-нейтральні імена (ті, що присутні кожного року)

Очікуваний результат:

Out[219]:

	0
0	James
1	Leslie
2	Joseph
3	Jessie
4	Jesse
5	Sidney
6	John
7	Robert
8	Tommie
9	Jean
10	Johnnie
11	William
12	Lee
13	Marion
14	Francis
15	Ollie

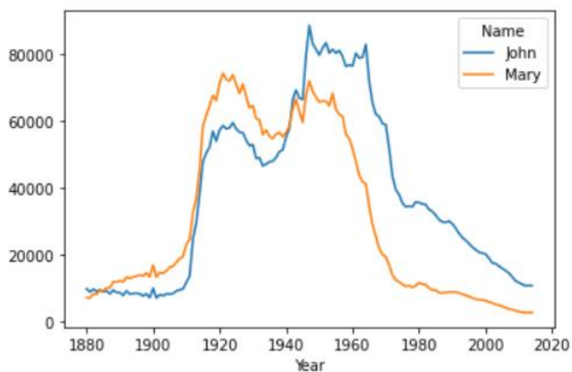
24. Знайти найпопулярніше серед непопулярних імен (непопулярне ім'я, яким називали дітей найбільшу кількість разів)

Очікуваний результат:

```
Out[94]: 'Наиболее популярное из непопулярных имен - это Celester. Им называли 160 раз'
```

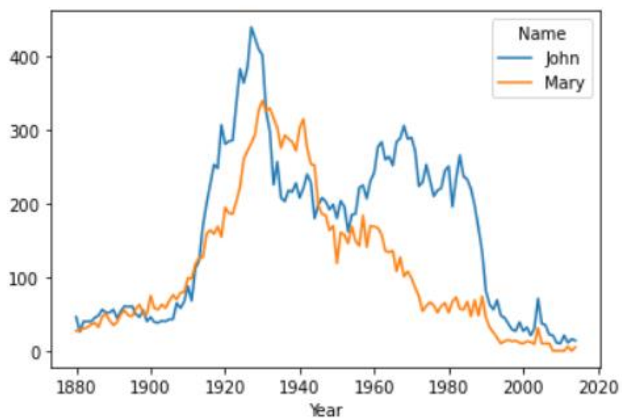
25. Побудувати графіки розподілення кількості імен John та Mary по роках без залежності до статі.

Очікуваний результат:



26. Побудувати графіки розподілення кількості жіночих імен John та чоловічих імен Mary по роках.

Очікуваний результат:



27. Знайти найпопулярніші імена в кожному році.

Очікуваний результат:

Out[214]:

Year	Name	Count
1880	John	9655
1881	John	8769
1882	John	9557
1883	John	8894
1884	John	9388
...	...	...
2010	Isabella	22883
2011	Sophia	21816
2012	Sophia	22267
2013	Sophia	21147
2014	Emma	20799

3. ЗМІСТ ЗВІТУ

1. Титульна сторінка звіту.
2. Тема та мета лабораторної роботи.
3. Хід роботи.
4. Посилання на створений блокнот Jupyter на GitHub.
5. Висновки.

Контрольні запитання

1. Чи потребує бібліотека Pandas окремого встановлення?
2. Назвіть та охарактеризуйте структури даних Pandas.
3. В чому збіжність та відмінність Pandas `Series` від одновимірного масиву `ndarray`?
4. В чому збіжність та відмінність Pandas `DataFrame` від двовимірного масиву `ndarray`?
5. Які властивості Pandas `Series` та `DataFrame` походять від стандартних словників Python?
6. Охарактеризуйте методи доступу до елементів з використанням явного та неявного індексів.

7. В чому полягає перевага використання атрибутів `loc` та `iloc`?

8. В чому сутність операції збереження індексу?

9. В чому сутність вирівнювання індексу?

10. Для яких об’єктів Pandas застосовуються операції вирівнювання та збереження індексів?

СПИСОК ЛІТЕРАТУРИ

1. Документація з Pandas. Режим доступу – http://pandas.pydata.org/

2. User Guide. Режим доступу – https://pandas.pydata.org/docs/user_guide/index.html

3. Методичні вказівки до лабораторної роботи «Основи роботи в середовищі Jupyter Notebook» з курсу «Обробка даних Python» [Електронний ресурс] : для студентів спец. 121 Інженерія програмного забезпечення, 122 Комп’ютерні науки, 124 Системний аналіз, 126 Інформаційні системи і технології / уклад.: С. М. Коваленко, С. В. Коваленко, О. В. Шматко ; Нац. техн. ун-т "Харків. політехн. ін-т". – Електрон. текст. дані. – Харків, 2021. – 28 с.

4. Методичні вказівки до лабораторної роботи «Основи роботи з бібліотекою NumPy» з курсу «Обробка даних Python» [Електронний ресурс] : для студентів спец. 121 Інженерія програмного забезпечення, 122 Комп’ютерні науки, 124 Системний аналіз, 126 Інформаційні системи і технології / уклад.: С. М. Коваленко, С. В. Коваленко, О. В. Шматко ; Нац. техн. ун-т "Харків. політехн. ін-т". – Електрон. текст. дані. – Харків, 2021. – 48 с.

Навчальне видання

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторної роботи

«Основи роботи з бібліотекою Pandas»

з курсу «Обробка даних Python»

для студентів спеціальностей 121 Інженерія програмного забезпечення,

122 Комп'ютерні науки, 124 Системний аналіз

Укладачі:

КОВАЛЕНКО Світлана Миколаївна

КОВАЛЕНКО Сергій Володимирович

Відповідальний за випуск Годлевський М. Д.

Роботу до видання рекомендував Безменов М. І.

В авторській редакції

План 2022, поз. 281

Підписано до друку 20.12.2022 р. Гарнітура Times New Roman.
Ум. друк. арк. 1,5.

Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2

Самостійне електронне видання