МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ФРАНКА

Роман Селіверстов
Андрій Мельничин

ОСНОВИ ПРОГРАМУВАННЯ МОВОЮ PYTHON

Навчальний посібник

Львів 2020

УДК 004.432
С 291

Рецензенти:

д - р. фіз. - мат. наук, доц. Нікітін А. В.
(Київський національний університет імені Тараса Шевченка);

канд. фіз. - мат. наук, Прищепа О. В.
(Національний університет водного господарства та природокористування);

д - р. тех. наук, проф. Соколовський Я. І.
(Національний лісотехнічний університет України).

Рекомендовано до друку Вченою радою Львівського національного університету імені Івана Франка. Протокол №85/5 від 25.05.2020 р.

Селіверстов Р. Г.

С 291 Основи програмування мовою Python : навчальний посібник / Селіверстов Р. Г., Мельничин А. В. – Львів : ЛНУ імені Івана Франка, 2020. – 190 с.

ISBN 978-617-10-0599-0

Послідовно та систематизовано викладено основні прийоми програмування мовою Python. З метою кращого засвоєння теоретичного матеріалу наведено достатню кількість прикладів та завдань для самоперевірки.

Для студентів факультету прикладної математики та інформатики, механіко-математичного факультету та усіх, хто цікавиться сучасними інформаційними технологіями.

УДК 004.432

ISBN 978-617-10-0599-0

ПЕРЕДМОВА

У сучасному інформаційному суспільстві комп'ютерні й інформаційні технології відіграють важливу роль в освіті, медицині, розважальному секторі, виробництві, комунікаціях, комерції, урядуванні та багатьох інших галузях. Щоб "змусити" машину виконувати щось, треба якось передати їй інструкції (написати програму).

Існує багато відмінних і водночас подібних між собою мов програмування, які класифікують за різними ознаками. Наприклад, за типом задач, які розв'язують (системні або прикладні мови; мови для розробки web-застосувань, організації баз даних, мобільних додатків тощо). Така ситуація стає зрозумілою, коли уявити кількість і різноманітність задач, які розв'язують засобами комп'ютерної техніки. Для цього потрібно використовувати різні інструменти, тобто різні мови програмування.

Багато розробників у минулому намагалися і, можливо, зараз намагаються ввести свою мову програмування, яка б мала переваги у зручності та якості розробки, порівняно з існуючими, у тій чи іншій ситуації.

Серед великої кількості популярних сьогодні мов програмування (Java, C++, PHP, C# тощо) ключовими вважаються лише кілька, зокрема і мова Python, яка успішно застосовується під час розробки графічних інтерфейсів, у системному програмуванні, веб-сценаріях, керуванні базами даних, наукових обчисленнях, програмуванні штучного інтелекту тощо.

Мова програмування Python завоювала чільне місце серед найпопулярніших мов програмування завдяки відкритості програмного забезпечення, читабельності та компактності коду, підтримці сучасних парадигм програмування, високій швидкості розробки, переносимості програм, обширній стандартній бібліотеці та підтримці бібліотек сторонніх розробників, інтегрованості з іншими мовами програмування.

На думку багатьох IT-фахівців, однією з заслуг Python є близький до псевдокоду[1] синтаксис, який робить мову легкою для вивчення початківцями. Водночас Python підтримує структури даних і об'єктно-орієнтоване програмування, тобто слугує потужним інструментом для програмістів-професіоналів.

Мета посібника – лаконічно викласти основи програмування на прикладі мови Python для зацікавлених осіб з мінімальним досвідом програмування, або які його не мають. У посібнику подано синтаксис, невеликі прості приклади програм і поради стосовно використання базових можливостей мови, але не вичерпна інформація стосовно притаманних їй функціональних можливостей (з такими можливостями можна ознайомитись у спеціалізованій літературі або документації Python). Також не проводиться порівняння з іншими мовами програмування – для тих, хто ніколи не програмував, таке порівняння було би малозрозумілим, а ті, хто має досвід програмування на інших мовах, самі зможуть побачити відмінності.

У кінці кожного розділу наведено питання та завдання для самоперевірки. Не на кожне з них є пряма відповідь у тексті розділу, тому рекомендуємо опрацьовувати додаткові джерела з тематики відповідного розділу.

Наприкінці посібника подано прокоментовані правильні відповіді на всі питання та розв'язки завдань. Щоб ліпше розуміти матеріал і вдосконалити практичні навички з програмування, варто набирати вручну та запускати наведені у посібнику фрагменти коду і приклади програм.

1 Псевдокод – неформальний запис алгоритму зрозумілою для людини мовою.

РОЗДІЛ 1
МОВА ПРОГРАМУВАННЯ PYTHON ТА ЇЇ ОСОБЛИВОСТІ

1.1. Загальні відомості про Python-програми

Python – багатоплатформна інтерпретована мова програмування високого рівня з відкритим кодом. Багатоплатформність мови Python полягає у здатності написаних на ній програм працювати на різних операційних системах і пристроях. На відміну від компільованих мов програмування, яким перед виконанням програми властиві етапи компіляції та компонування, Python як інтерпретована мова виконує код одразу[2]. Належність до класу високорівневих мов програмування означає, що програмісту не потрібно глибоко вникати в особливості функціонування апаратного забезпечення й операційних систем – запис команд нагадує англійську мову, тому зрозумілий. Відкритість коду дає змогу користувачам вільно переглядати код і використовувати його для створення своїх програм.

Головним недоліком мови Python вважають її невисоку, порівняно з компільованими мовами програмування, продуктивність. Проте для більшості застосунків такої продуктивності цілком достатньо. Якщо ж питання швидкості виконання програми принципово важливе, то окремі її частини реалізують у вигляді розширень на мові C.

Програму можна трактувати як подання на певній мові програмування деякого алгоритму – послідовності чітко визначених інструкцій для виконання певного завдання (отримання бажаного результату). Класична програма складається з трьох частин (блоків): отримання вхідної інформації (даних), обробки даних за певним алгоритмом і видачі результату. Для написання програми на мові Python достатньо будь-якого текстового редактора, але для її запуску (виконання) на комп'ютері має бути встановлена програма-інтерпретатор

2 Насправді згадані етапи приховані від користувача та виконуються інтерпретатором автоматично.

Python, яка постачається разом з інтегрованим середовищем розробки IDLE. Крім IDLE, є й інші середовища розробки (Eclipse, NetBeans, PyCharm, Geany, Jupyter Notebook тощо) та веб-ресурси для онлайнового запуску програм (trinket.io, jdoodle.com та ін.)

Наразі співіснують дві паралельні версії (гілки) мови – Python 2 і Python 3. Хоча версія Python 2.7 ще деякий час буде підтримуватися, розробники Python рекомендують переводити існуючі на ній проекти на третю версію. Тому саме її розглядатимемо у цьому посібнику.[3]

1.1.1. Структура програми

Попри те, що теоретично код програми може міститися в одному файлі, для більшості реальних програм він записаний у багатьох файлах, які називаються модулями. Кожен модуль містить інструкції, які складаються з виразів. Вирази створюють різноманітні об'єкти й оперують ними.

1.1.2. Коментування коду

Щоб зробити код програми зрозумілішим, рекомендовано використовувати коментарі – фрагменти коду з пояснювальними дописами, які не виконуються (пропускаються) інтерпретатором[4]. Python дає змогу додавати однорядкові та багаторядкові коментарі.

Однорядковий коментар розпочинається символом **#**. Усе, що слідує за цим символом до кінця рядка, вважається коментарем і не впливає на виконання програми. Багаторядковий коментар починається і закінчується трьома символами `'''` або `"""`. Нижче наведено зразок програми *1.py* до якої додано коментарі:

```
'''Програма обчислює квадратний корінь з числа, записаного у
змінну data, та виводить отриманий результат на екран'''
from math import sqrt    # імпорт функції для обчислення кореня
```

3 На момент написання це версія Python 3.7.1.

4 Коментарі також часто використовують для тимчасового "відключення" частини коду.

```
data = 100           # число, з якого потрібно добути корінь
result = sqrt(data)  # добування кореня
# --- Друк результату: ---
print('Результат: ', result)
```

Наприклад, із коментаря `# --- Друк результату: ---` стає зрозумілим, що для виведення інформації на екран потрібно ввести `print`, після чого у круглих дужках через кому перерахувати, що саме треба відобразити. Запам'ятайте наразі, що послідовність символів, взята в одинарні або подвійні лапки, називається рядком і виводиться на екран у такому ж вигляді.

1.1.3. Інтерактивний режим

Після запуску IDLE за замовчуванням відкривається вікно командної оболонки Python – інтерактивний режим, у якому інструкція виконується одразу після її введення. Python запрошує користувача до введення інструкції послідовністю символів `>>>` на початку рядка. Ввівши інструкцію або вираз та натиснувши клавішу *Enter*, користувач отримує у наступному рядку результат і нове запрошення[5]. Приклад сеансу роботи в інтерактивному режимі зображено на рис. 1.1.

Перейти до інтерактивного сеансу роботи можна також і безпосередньо з командного рядка (консолі) операційної системи, ввівши команду `python3` без аргументів.

Інтерактивний режим використовується як допоміжний під час написання програм. У ньому можна швидко подивитись результати виконання коротких виразів та інструкцій, а також почитати документацію, ввівши `help()`. Ніщо зі зробленого в інтерактивному сеансі не зберігається після виходу з нього. Для збереження програмного коду використовується сценарний режим.

5 Не всі інструкції виконуються одразу після натискання клавіші Enter. Python підтримує складені (багаторядкові) інструкції, які розглядатимуться пізніше.

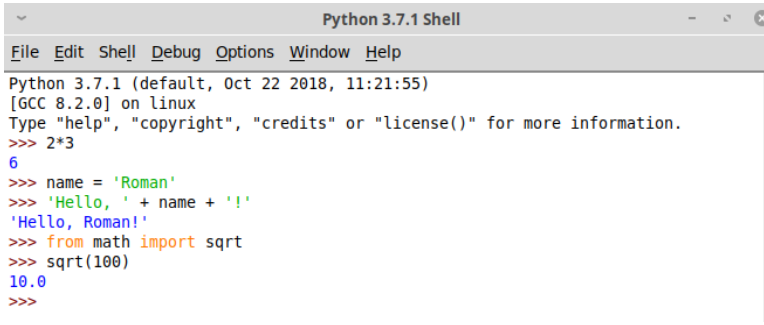


Рис. 1.1. Приклад сеансу роботи в інтерактивному режимі

1.1.4. Сценарний режим

Під час роботи у сценарному режимі програма записується у файл з розширенням .py (скрипт). Для переходу до сценарного режиму використовується команда *New File* з меню *File*. Вона відкриває вікно текстового редактора, в якому друкується текст програми. Для виконання програми потрібно виконати команду *Run Module* з меню *Run* (або натиснути клавішу *F5*). За замовчуванням інтерпретатор не виконуватиме незбережену програму, отож, якщо ви не зберегли її перед тим, як виконати, то з'явиться відповідне повідомлення. Результат виконання програми буде відображено у вікні інтерактивного режиму.

Не вдаючись у деталі коду, нижче продемонстровано текст програми 1.py[6] для обчислення квадратного кореня з числа 100 (рис. 1.2) і результат її виконання (рис. 1.3).

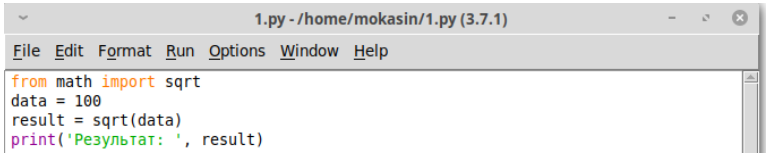


Рис. 1.2. Вікно з текстом (кодом) програми

6 Починати імена файлів з цифри не заборонено, але не рекомендовано, бо такі файли не зможуть за потреби імпортуватися.

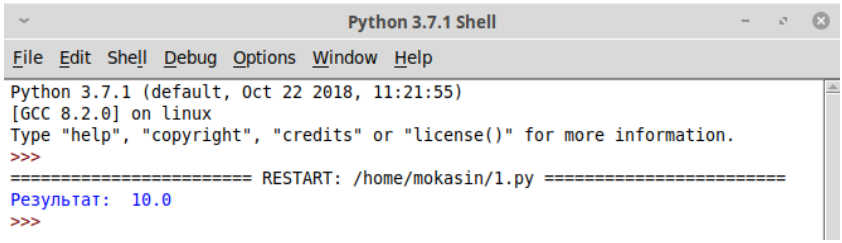


Рис. 1.3. Вікно з результатом виконання програми

Символи `>>>` в останньому рядку вікна командної оболонки засвідчують завершення виконання програми. Якщо вони не з'являються впродовж тривалого часу[7], то програму можна примусово зупинити комбінацією клавіш *Ctrl+C*.

Запустити програму можна і з командного рядка[8] операційної системи. Для цього потрібно передати ім'я файлу як аргумент команді[9] `py -3` (у Windows)

```
py -3 1.py
```

або `python3` (в інших операційних системах):

```
python3 1.py
```

Додатково можна вказати назву файлу, у який запишуться результати виконання програми:

```
python3 1.py > result.txt
```

Детальніше з цими й іншими способами та особливостями запуску python-програм можна ознайомитись у спеціалізованій літературі.

---

7 Причин може бути багато: програма очікує на введення даних, зациклювання, алгоритм обчислювально складний (потребує великої кількості обчислень) тощо.

8 Робота в командному рядку (командній оболонці) в посібнику не розглядається.

9 Команди мають виконуватися з папки, у якій збережений файл програми. У протилежному випадку замість імені файлу треба записати повний шлях до нього.

1.2. Налагодження програми

Програми рідко спрацьовують з першого разу. Під час їх написання трапляються помилки, адже програми пишуть люди. Розрізняють синтаксичні помилки, семантичні помилки та помилки виконання. Процес виявлення, знаходження (локалізації) та виправлення помилок програмування називається відлагодженням. Спеціальна програма для пошуку помилок у програмному коді називається відлагоджувачем. Стандартна бібліотека Python містить відлагоджувач командного рядка pdb. Більшість інтегрованих середовищ розробки мають свої відлагоджувачі. Початківцям у програмуванні рекомендується найпростіший спосіб відлагодження програми, який полягає у виводі результату після виконання кожної інструкції.

1.2.1. Синтаксичні помилки

Python має чітко визначений синтаксис. Це означає, що кожна інструкція має бути записана згідно з визначеними правилами. Якщо у тексті програми хоча б одне з таких правил буде порушено, то у разі спроби виконати програму з'явиться повідомлення про синтаксичну помилку (syntax error). Допоки усі синтаксичні помилки не будуть виправлені, інтерпретатор не виконуватиме програму. Наприклад, спробуйте перший рядок наведеної вище програми 1.py замінити на рядок ніби-то такого ж (з погляду лінгвістики) змісту:

```
import sqrt from math
```

Спроба запустити програму буде невдалою – після натискання *F5* з'явиться випливаюче вікно з повідомленням про синтаксичну помилку "invalid syntax". Річ у тому, що ми порушили синтаксис Python, помінявши місцями інструкції `import` і `from`.

Подібне повідомлення з написом `"unexpected indent"` (неочікуваний відступ) з'явиться у випадку, коли інструкцію почати не з початку рядка, а поставити перед нею один чи кілька пробілів – відступи є складовою синтаксису мови Python і їхнє використання строго регламентоване. На цьому етапі (поки не розглядалися складені

інструкції) треба запам'ятати, що кожна інструкція має починатися з першого символу рядка, тобто без жодного відступу.

Синтаксичні помилки можуть трапитися і у разі повного дотримання синтаксису – тоді повідомлення про них (з зазначенням файлу, рядка і типу помилки) відображаються в командній оболонці. Наприклад, якщо допустити описку у назві змінної у третьому рядку:

```
result = sqrt(date)
```

то отримаємо повідомлення про помилку, яке свідчить про те, що інтерпретатор не може добути квадратний корінь, оскільки не знає значення `date`:

```
Traceback (most recent call last):
  File "/home/mokasin/1.py", line 3, in <module>
    result = sqrt(date)
NameError: name 'date' is not defined
```

1.2.2. Помилки виконання

Помилки виконання (runtime errors) виникають під час виконання синтаксично правильно написаної програми. Оскільки вони з'являються лише в окремих випадках, то їх також називають винятками (exceptions)[10]. Якщо ми спробуємо добути корінь з від'ємного числа (наприклад, `data = -100`), то отримаємо повідомлення саме про таку помилку.

1.2.3. Семантичні помилки

Семантичні помилки (semantic errors) пов'язані з неправильною реалізацією алгоритму. Програма за наявності таких помилок виконуватиметься, але видаватиме неправильний результат. Наприклад, якщо у третьому рядку "забути" викликати функцію добування кореня:

```
result = data
```

10 Опрацюванню винятків присвячений розділ 9.

то інтерпретатор не побачить такої помилки і виконає програму. Щоправда, результат буде не той, який очікували (100 замість 10.0).

На практиці локалізовувати семантичні помилки доволі складно, бо інтерпретатор не дає для них жодних підказок.

Питання та завдання для самоперевірки

1. Назвати головний недолік Python порівняно з компільованими мовами програмування.
2. В інтерактивному режимі виконано присвоєння значення змінній `sto`:
   ```
   >>> sto = 10 * 10
   ```
   Яким буде значення змінної `sto`? Чи буде воно виведене на екран? Якщо ні, то як його відобразити в інтерактивній оболонці?
   В інтерактивному режимі введено команду
   ```
   >>> 100 / (50 - 2 * 25)
   ```
   Повідомлення про який тип помилки з'явиться?
3. Поміркувати, які складові коду програми з рис. 2 можна змінити на щось інше (використати інші імена для них), а які мають залишитися незмінними.

РОЗДІЛ 2
БАЗОВІ ПОНЯТТЯ ТА ТЕРМІНОЛОГІЯ

2.1. Основні терміни та поняття

2.1.1. Типи даних

Будь-які дані, які використовуються комп'ютерною програмою, належать до певного типу – числа, літери, дати тощо. Належність до конкретного типу визначає сукупність операцій, які можна виконувати над даними. Наприклад, над числами можна виконувати арифметичні операції, для літер можна змінювати регістр, дати можна відображати у різних форматах тощо. Тобто, тип даних визначається множиною всіляких можливих значень даних цього типу та множиною всіляких можливих операцій, які можуть виконуватися над ними.

Код Python-програми містить послідовність інструкцій. Кожна інструкція складається з виразів, які створюють об'єкти та виконують операції над ними. Python має вбудовані (постійно доступні, готові до використання) об'єкти та інструменти для створення нових. До базових вбудованих типів об'єктів належать: числа, рядки, списки, словники, кортежі, множини, файли, функції та ін. Для роботи з кожним типом Python пропонує набір операцій, які реалізуються з використанням операторів, функцій, методів та ін.

2.1.2. Літерали

Літерал – постійне значення певного типу даних, записане у коді програми. Літерал можна розглядати як форму запису даних, за якою інтерпретатор однозначно визначає тип і значення цих даних. Наприклад, `50` – літерал цілого числа п'ятдесят, `'моя програма'` – літерал рядка з 12-ти символів (пробіл теж символ!), `[]` – літерал порожнього списку тощо.

2.1.3. Вирази й оператори

Вирази забезпечують виконання операцій над даними для продукування нових даних. Оператори – невід'ємна складова виразів. Наприклад, у виразі `A + B` оператором є символ "+". Інтерпретатор розуміє цей вираз як команду застосувати оператор `+` до значень `A` і `B` та повернути результат цього виразу. Як саме обчислюватиметься значення виразу і що буде його результатом залежить від типу значень у виразі. Тобто, той самий оператор може виконувати різні операції залежно від типу значень, до яких він застосовується (ця властивість називається поліморфізмом). Наприклад, для чисел оператор `+` виконує арифметичну дію додавання, а для рядків – їх конкатенацію:

```
print(2 + 3)                # 5[11]
print('2' + '3')            # 23
```

Спроба застосувати оператор `+` до значень різних типів призведе до помилки виконання:

```
print('2' + 3)
# TypeError: can only concatenate str (not "int") to str
print(2 + '3')
# TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Для прикладу нижче наведено далеко не повний перелік операторів Python (зміст та особливості застосування цих та інших операторів будуть розглянуті поступово у наступних розділах):

Арифметичні оператори: `+, -, *, /, **, //, %` та ін.
Логічні оператори: `and, or, not.`
Оператори порівняння: `>, <, >=, <=, ==, !=.`
Оператори перевірки на входження: `in, not in.`
Оператор присвоєння: `=.`

11 Тут і надалі у коментарі рядка (або після нього) з функцією print() відображається результат її виконання.

Кілька операторів можна використовувати в межах одного виразу. У цьому випадку інтерпретатор виконуватиме операції у такому змішаному виразі за порядком старшинства операторів:

```
print(2 + 2 * 2)              # 6
```

Для зміни порядку старшинства використовуються круглі дужки:

```
print((2 + 2) * 2)            # 8
```

Пробіли всередині виразів не мають принципового значення (усі вони ігноруються інтерпретатором) – їх використовують лише з метою поліпшення читабельності коду.

Складні (довгі) вирази можна записувати у кілька рядків, поставивши перед переходом на новий рядок символ оберненої косої риски:

```
sum_10 = 1 + 2 + 3 + 4 + 5 + \
         6 + 7 + 8 + 9 + 10
```

або охопивши вираз круглими дужками.

2.1.4. Інструкції

Інструкції – логічні блоки програми, які містять вирази і забезпечують виконання конкретних команд. Прикладами інструкцій є: `if/elif/else` (операція вибору), `break` (вихід з циклу), `def` (створення функції), `import` (доступ до модулів) тощо.

Кінцем простих інструкцій слугує кінець рядка, тобто кожна проста інструкція записується в одному рядку. Python допускає запис кількох простих інструкцій в одному рядку, розділяючи їх крапкою з комою:

```
a = 2; print(a**2)            # 4
```

але такий стиль непритаманний Python-програмам.

Якщо інструкція містить вкладені інструкції, то вона називається складеною і записується у кілька рядків, використовуючи такий синтаксис:

```
Основна_інструкція:
    Блок_вкладених_інструкцій
```

Тобто, в кінці основної інструкції ставиться двокрапка, а кожна вкладена інструкція розпочинається з обов'язкового однакового відступу, який за замовчуванням складається з чотирьох пробілів або одного символу табуляції.

Наприклад, один з варіантів реалізації умовної інструкції виглядає так:

```
if hours > 24:
    print('Пройшло більше доби')
else:
    print('Пройшло менше доби')
```

Якщо вкладена інструкція теж є вкладеною інструкцією, то відступи задаються уже стосовно неї:

```
if hours > 24:
    if hours > 48:
        print('Пройшло більше двох діб')
    else:
        print('Пройшло більше доби')
else:
    print('Пройшло менше доби')
```

Відступи є одним з таких елементів синтаксису, який відрізняє Python від більшості інших мов програмування. Вони поліпшують читабельність програми і є по суті єдиним механізмом, який надає Python для групування коду. Для організації відступів, крім пробілів, можна використовувати символ табуляції. Але не допускається змішування пробілів і символів табуляції в межах одного блоку – інтерпретатор видасть синтаксичну помилку "unexpected indent".

Якщо фрагмент коду програми міститься всередині дужок (круглих, квадратних або фігурних), то він може бути записаний у кілька рядків попри те, що належить одній простій інструкції. Наприклад,

```
sum_10 = (1 + 2 + 3 + 4 + 5 +
          6 + 7 + 8 + 9 + 10)
days = ["Понеділок", "Вівторок",
```

```
"Середа", "Четвер", "П'ятниця",
"Субота", "Неділя"]
```

2.1.5. Змінні

Для полегшення запам'ятовування літералів і їхнього подальшого використання у коді програми використовуються змінні. В останньому фрагменті коду результат підсумовування перших десяти натуральних чисел збережено у змінну з іменем `sum_10`, а список днів тижня у змінну `days`. Імена змінних можна вибирати довільно, але є кілька обмежень:

1) не можна як ім'я змінної використовувати зарезервовані імена, які ще називають ключовими словами Python:

```
False       None          True        and          as
assert      asyncawait    break       class        continue
def         del           elifelse    for          finally
from        global        ifimport    lambda       nonlocal
passreturn  not           try         while        with
```

Для детальнішої інформації щодо зарезервованих імен варто скористатися командою `help> keywords;`

2) ім'я змінної має починатися з літери або символу підкреслення "_" і може складатись з довільної кількості літер, цифр і символів підкреслення (використання інших символів заборонене).

Крім того, регістр літер теж має значення: `somename`, `Somename`, `someName`, `SOMENAME` і т. д. – усе це різні змінні.

У колах програмістів для імен змінних прийнято використовувати літери нижнього регістру, а якщо ім'я містить кілька слів, то усі слова, крім першого, починати з великої літери, наприклад `speed`, `maxSpeed`, `myMaxSpeed`[12]. Також можна іменувати змінні за допомогою малих літер і символів підкреслення (наприклад, `max_speed`, `my_max_speed`), але цей спосіб вважається пріоритетним для імен функцій і методів. Якщо це не

12 Рекомендації стосовно оформлення коду зібрані в документі PEP 8, доступному за посиланням www.python.org/dev/peps.

заважає читабельності, то ліпше використовувати імена тільки з літер нижнього регістра: `maxspeed, mymaxspeed`.

Звичайно, не забороняється назвати змінні `MaxSpeed` чи `Max_Speed`, але перший варіант зарезервований для іменування класів, а другий взагалі бажано не використовувати.

Якщо значення змінної залишається незмінним впродовж виконання усієї програми (такі змінні називаються константами), то їх іменують великими літерами, а для розділення слів використовують символ підкреслення: `SIZE, STANDARD_SIZE`.

Для того, щоб асоціювати конкретне значення літералу зі змінною, використовується оператор присвоєння `=`. Синтаксис операції присвоєння такий:

```
<ім'яЗмінної> = <вираз>
```[13]

Інтерпретатор спочатку обчислить значення виразу справа від оператора присвоєння, після чого асоціює це значення з іменем змінної, зазначеним зліва.

Значення літерала (виразу) після створення (обчислення) записується в певне місце пам'яті, адресу якого можна отримати за допомогою функції[14] `id()`, наприклад,

```
print(id(7))          # 10968992
```

Змінні дають змогу програмісту замість складних числових адрес запам'ятовувати їхні сприйнятливіші та змістовніші словесні відповідники, адже імена змінних є по суті вказівниками на адреси, за якими зберігаються їхні значення в пам'яті комп'ютера:

```
x = 7
print(id(x))          # 10968992
```

Для прикладу наведемо програму для обчислення довжини кола, яка використовує три змінні (`pi, radius, l`):

---

13 Тут і надалі в дужках <> позначені елементи коду, які мають заміщуватися іменами або виразами користувача.

14 Поняття функції детальніше розглядатиметься у розділі 8.

```
pi = 3.        # записуємо значення числа “пі” у змінну pi
radius = 1     # записуємо значення радіусу у змінну radius
l = 2 * pi * r
# обчислюємо довжину кола і записуємо у змінну l
print(l)       # виводимо на екран значення змінної l
```

Не можна використовувати змінну у виразах, якщо раніше цій змінній не було присвоєно значення. Якщо попередня програма набуде вигляду

```
r = 1
l = 2 * pi * r
pi = 3.14
print(l)
```

то, оскільки на момент обчислення довжини кола (другий рядок) інтерпретатору Python нічого не відомо про значення змінної pi (яке присвоюється у наступному рядку), з’явиться повідомлення про помилку[15]:

```
NameError: name 'pi' is not defined
```

Якщо кільком змінним треба надати однакове значення, то використовують групове присвоєння[16], наприклад,

```
var1, var2, var3 = 0  # трьом змінним присвоєно значення 0
```

У Python змінні починають своє існування в момент їхнього першого використання. У цьому випадку тип змінних (на відміну від багатьох інших мов програмування) не потребує попереднього оголошення, а визначається автоматично під час виконання (така модель називається динамічною типізацією). Тобто, тип змінної визначається

15 Тут і надалі наводиться тільки останній рядок повідомлення про помилку.

16 Групове присвоєння використовують зазвичай для присвоєння значень незмінюваних типів (числа, рядки, кортежі). Присвоєння значень змінюваних типів (списків, словників) призведе до того, що зміна значення однієї змінної спричинить зміну значень інших змінних (див. підпункт 7.1.7.).

типом об’єкта, який присвоюється цій змінній. Більше того, тип змінної може змінюватися в межах одного скрипту (програми):

```
v = '10'        # змінна посилається на об’єкт типу str (рядок)
v = 10          # а тут на об’єкт типу int (ціле число)
v = v/2         # а тепер на об’єкт float (дійсне число)
```

У попередньому фрагменті коду змінна з іменем `v` створюється в момент присвоєння їй значення `10` типу `int`. Подальші присвоєння просто змінюють асоційоване з іменем змінної значення. Якщо висловлюватися коректно, то змінні не мають типу (не несуть ніякої інформації про тип), а є лише посиланнями на об’єкти певного типу в певний момент часу. У виразах змінні заміняються значеннями, на які вони посилаються (у третьому рядку коду спочатку обчислилось значення виразу `v/2`, у якому замість змінної `v` підставилось її поточне значення `10`, а потім це значення (дія ділення завжди повертає дійсний тип `float`) було знову асоційоване зі змінною `v`).

2.1.6. Функції

Функцію в Python можна сприймати як іменований фрагмент коду, який виконує певні операції. Функція може мати аргументи – вхідні дані, від значень яких залежить результат її виконання. Для виклику функції потрібно після її імені у круглих дужках через кому зазначити необхідні значення аргументів (якщо функція не має аргументів – круглі дужки залишають порожніми)[17]. Кожен аргумент має певний тип даних – якщо функції передати аргумент іншого (не очікуваного функцією) типу, то інтерпретатор згенерує повідомлення про помилку виконання.

У попередніх фрагментах коду вже використовували функції, зокрема `print()` і `sqrt()`. Проаналізувавши їхнє використання, можна дійти висновку, що результатом виконання функції є не тільки виконання певних дій (як у випадку функції `print()`, яка виводить на екран значення

17 Детальніше про способи задання агрументів, виклики та інші особливості функцій описано далі.

аргументів) – функція може також повертати значення (функція `sqrt()` повертала значення квадратного кореня, яке записувалося у змінну `result`).

Отже, загальний синтаксис виклику функції такий:

```
<ім'я_функції>(<аргумент1>, <аргумент2>,..., <аргументN>)
```

Якщо аргументом функції є літерал або змінна, то функції передається значення цього літерала (змінної). Аргументом функції може бути вираз – тоді спочатку обчислюється його значення, а потім це значення передається функції як аргумент. У цьому випадку вираз може містити виклики інших функцій.

Приклад виклику функції:

```
max(-1, 0.25, 1/8)    # повертає найбільше число, тобто 0.25
```

У Python є величезна кількість функцій. Деякі з них вбудовані, інші потребують імпортування модулів стандартної бібліотеки або навіть інсталяції зовнішніх бібліотек. Значна частина функцій стандартної бібліотеки Python розглядатиметься у подальших розділах цього посібника.

Якщо функція є частиною класу, то вона називається методом. Методи своєю поведінкою нагадують функції, але мають інший синтаксис виклику, який використовує точкову нотацію:

```
<об'єкт>.<метод>(<аргумент1>, <аргумент2>, ..., <аргументN>)
```

Приклад виклику методу:

```
"1123101".count('1')  # повертає кількість одиниць, тобто 4
```

2.1.7. Об'єкти та класи

В Python все (числа, рядки, списки, словники, кортежі, множини, файли, модулі, функції, класи) є об'єктами і можуть бути присвоєні змінній. Стан об'єктів описується їхніми атрибутами, а поведінка – методами. Класи дають змогу створювати користувацькі об'єкти з особливими станом і поведінкою.

2.2. Засоби вводу-виводу інформації у Phyton

2.2.1. Функція виводу print()

Функція `print()` використовується для виведення (відображення) інформації на екран[18]. У дужках через кому зазначають аргументи (вирази або змінні, значення яких мають бути виведені), наприклад,

```
a = 2 + 3
print('2 + 3 =', a)                  # 2 + 3 = 5
print('2 * 3 =', 2 * 3)              # 2 * 3 = 6
```

Літерали рядків відображаються на екрані без охоплюючих лапок, якщо аргументом є змінна чи вираз, то відображатимуться значення цієї змінної чи виразу, відповідно.

За замовчуванням значення аргументів виводяться на екран через пробіл, а функція `print()` завершується символом кінця рядка, тобто кожне виведення розпочинатиметься з нового рядка.

За допомогою аргументу `sep` можна використати довільний розділювач:

```
a = 2 + 3
s = '2 + 3'
print(s, a, sep=' = ')               # 2 + 3 = 5
```

Якщо потрібно, щоб вивід закінчувався іншими символами, то треба зазначити їх як значення аргументу `end`:

```
print('2 + 3 =', 2 + 3, end='; ')
print('2 * 3 =', 2 * 3)
# на екран виведеться один рядок: 2 + 3 = 5; 2 * 3 = 6
```

Аргументи `sep` і `end` можна використовувати разом:

```
print('2 + 3', 2+3, sep=' = ', end='; ')
```

18 В інтерактивному режимі ф-ю print() явно використовувати не обов'язково, бо значення виразу чи змінної виводиться автоматично після натискання клавіші Enter.

```
print('2 * 3', 2*3, sep=' = ', end='.')
# на екран виведеться один рядок: 2 + 3 = 5; 2 * 3 = 6.
```

Гнучкіші способи виведення інформації розглянемо далі.

2.2.2. Введення даних за допомогою функції input()

Функція `input()` використовується для введення значення з клавіатури. Зазвичай у дужках зазначають рядок з пояснювальним текстом. Під час виклику функція виводить на екран пояснювальний текст і чекає, доки користувач не введе значення з клавіатури. Після натискання клавіші *Enter* функція приймає і одразу повертає це значення, після чого виконання програми продовжується. У більшості випадків введене значення записується у змінну, наприклад,

```
n = input('Введіть число: ')
```

Проте, якщо ми захочемо, наприклад, збільшити це число на 1:

```
n += 1          # Те саме, що n = n + 1
```

то під час виконання програми трапиться помилка, навіть якщо ми введемо з клавіатури ціле число:

```
TypeError: can only concatenate str (not "int") to str
```

Річ у тім, що усе введене з клавіатури сприймається інтерпретатором як рядок (тип `str`), а не як число – для перетворення у числовий тип потрібно скористатися функцією `int()` або `float()`:

```
n = input('Введіть число: ')
n = int(n)
```

Це можна записати й одним рядком:

```
n = int(input('Введіть число: '))
```

Тоді програма “Наступне ціле число” виглядатиме так:

```
n = int(input('Введіть число: '))
n += 1
print('Наступне число:', n)
```

або у два рядки:

```
n = int(input('Введіть число: '))
print('Наступне число:', n+1)
```

Цю програму можна записати й у один рядок, але такі записи не завжди доцільні, бо їх складно читати:

```
print('Наступне число:', int(input('Введіть число: '))+1)
```

Питання та завдання для самоперевірки

1. Якщо деякий рядок розпочинається відступом, то чи означає це, що попередній рядок закінчується двокрапкою?
2. Що з наведеного можна використати як ім'я змінної? Чому?
   `1st_day or Day_1 day-1 ro`
3. Викликати функцію з іменем `some_func`, передавши їй аргументом значення, асоційоване зі змінною `value`.
4. Застосувати до об'єкта, асоційованому зі змінною `value`, метод з іменем `some_method` без аргументів.
5. Чому синтаксично правильний код не видає на екран бажаний результат (число 8)?
   ```
   a = 1; b = 2; c = 3
   (a + c) * b
   ```
6. Що надрукує функція `print()`, якщо їй не передати жодного аргументу?
7. Чи можна за допомогою функції `print()` вивести (записати) інформацію у файл?
8. Написати в один рядок програму для обчислення подвоєння введеного з клавіатури дійсного числа.

РОЗДІЛ 3
ЧИСЛОВІ ТИПИ ДАНИХ МОВИ PYTHON

До вбудованих числових типів Python належать цілі, дійсні та комплексні числа. Стандартна бібліотека Python (див. розділ “Стандартна бібліотека Python”) містить також модулі для роботи з числами з фіксованою точністю (`decimal`) і раціональними дробами (`fractions`).

3.1. Цілі числа

Літерали цілих чисел (тип `int`) записуються послідовністю десяткових цифр, перед якими може стояти знак числа (додатні числа зазвичай записуються без знака): `125, -13, +64, 0`. Python дає змогу подавати числа у двійковій (починаються з префікса `0b`), вісімковій (`0o`) і шістнадцятковій (`0x`) системах числення. Наприклад, літерал числа `111` у різних системах числення записується так: `111, 0b1101111, 0o157, 0x6f`.

Python підтримує необмежену точність подання цілих чисел[19]. Наприклад, результатом виразу `2**100000` (два у стотисячному степені) буде число, яке складається зі `30103` цифр.

3.2. Дійсні числа

Літерали дійсних чисел (тип `float`[20]) розпізнаються інтерпретатором за символом десяткової крапки і/або експоненти (символ “`e`” або “`E`”): `3.14, 2.0, 2.` (те саме, що `2.0`), `.2` (`0.2`), `1.5e-2` (`0.015`), `4E3` (`4000.0`, а не `4000`), `5.e+2` (`500.0`). Діапазон значень типу `float` – приблизно від -10^{308} до 10^{308} з 16-ма десятковими знаками.

19 Кількість цифр обмежується лише обсягом доступної пам’яті.

20 Насправді тип float – це так звані числа з плаваючою крапкою, які не завжди дорівнюють дійсному числу, яке вони репрезентують.

3.3. Комплексні числа

Комплексні числа розпізнаються інтерпретатором за символом уявної одиниці (“j” або “J”), який ставиться в кінці уявної частини: `3+2j`, `3.-2.5J`, `6j`. Їх можна також створити за допомогою вбудованої функції `complex(real, imagine)`, аргументами якої є дійсна `real` та уявна `imagine` частини.

Незалежно від запису (`2+j` чи `2.+1.j`) дійсна та уявна частини є дійсними числами (`2.0` і `1.0`, відповідно). Отримати окремо дійсну та уявну частину можна через атрибути `real` та `imag` комплексного числа:

```
print((2 + j).real)          # 2.0
c = 2 + j
print(c.imag)                # 1.0
```

3.4. Логічний тип

Логічний тип (`bool`) складається з двох взаємовиключних значень, які можуть бути записані кількома способами, але найуживанішими є пара `True` і `False`[21]. Вони трактуються як “правда” і “неправда”, “так” і “ні” тощо[22]. В Python логічний тип є підмножиною типу `int`, оскільки його значення поводяться так само, як цілі числа `1` і `0`, лише відображаються як `True` і `False` для ліпшого сприйняття:

```
print(True + False)          # 1
```

3.5. Операції з числовими типами даних

Використовуючи арифметичні оператори, над числами можна виконувати такі операції (у дужках зазначено відповідний оператор):

21 Великі літери важливі, інакше інтерпретатор трактуватиме true і false як імена змінних.

22 False відповідає числу 0, значенню None, порожньому рядку і порожній структурі даних.

додавання (+), віднімання (-), множення (*), ділення (/), отримання цілої частини (//) й остачі (%) від ділення, піднесення до степеня (**) та ін.

Наведений нижче код демонструє застосування цих операторів над цілими числами:

```
a = 123
b = 10
print(-a, a + b, a – b, a * b, a / b)  # -123 133 113 1230 12.3
print(a ** b)                          # 792594609605189126649
print(a // b, a % b)                   # 12 3
```

Результатом ділення завжди буде дійсне число. Якщо хоча б одне з чисел у виразі дійсне, то результатом виразу також буде дійсне число:

```
print(123 / 3, 123. // 10)             # 41.0 12.0
```

У складених виразах пріоритетність операторів така: піднесення до степеня, зміна знака, множення та усі види ділення, додавання та віднімання. Для зміни черговості операцій (групування підвиразів) використовуються круглі дужки:

```
print(3*-5**2)                         # -75
print((3*-5)**2)                       # 225
```

3.6. Вбудовані функції та методи для роботи з даними

Для операцій над числами, крім розглянутих вище операторів, можна користуватися вбудованими функціями. Розглянемо деякі з них.

Функції конвертування типів `int()` і `float()` можуть приймати аргументом число або рядок:

```
int(3.95)      # 3 (відтинання дробової частини дійсного числа)
float(3)       # 3.0 (перетворення цілого числа в дійсне)
int('3')       # 3
float('3')     # 3.0
int('3.95')
# ValueError: invalid literal for int() with base 10: '3.95'
```

```
float('3.95')         # 3.95
float('3,95')
# ValueError: could not convert string to float: '3,95'
```

Функції переведення в інші системи числення:

```
bin(111)              # '0b1101111' (у двійкову)
oct(0b1101111)        # '0o157' (у вісімкову)
hex(0o157)            # '0x6f' (у шістнадцяткову)
```

Зауважте, що результатом виконання цих функцій є рядок (тип `str`), а не число (тип `int`). Тобто, вони видають лише символьний запис числа в іншій системі числення.

Функція заокруглення `round()` приймає першим аргументом число, а другим (необов'язковим) – кількість десяткових знаків, до якої треба його заокруглити:

```
round(3.95)       # 4 (заокруглення до найближчого цілого)
round(3.95, 1)    # 4.0
round(1/3, 3)     # 0.333
abs(-2)           # 2 (повертає модуль числа, комплексного теж)
pow(2, 5)         # 32 (підносить 2 до 5-ї степені)
```

Крім того, числові типи як об'єкти мають різноманітні методи. Наприклад, дійсні числа мають метод перетворення у раціональне число (повертає кортеж з двох значень – чисельника та знаменника):

```
4.5.as_integer_ratio()       # (9, 2)
```

`ratio()` – метод обчислення кількості бітів, потрібних для їхнього подання:

```
n = 2500
n.bit_length()               # 12
```

а комплексні – метод обчислення комплексно спряженого числа:

```
2+3j.conjugate()             # (2-3j)
```

Питання та завдання для самоперевірки

1. Не використовуючи функції `pow()`, побудувати вираз, значенням якого є корінь квадратний з числа `x`.
2. На прикладі довільного комплексного числа переконатися, що множення цього числа на комплексно спряжене рівносильне обчисленню квадрата модуля цього числа.
3. Написати програму, яка за введеним часом (змінні `hours`, `minutes` і `seconds`) і числом `n` повертає час, який буде через `n` секунд від заданого.
4. Яке з наведених чисел типу `float` зайве?
   ```
   1320.
   132e+2
   132E-1
   1.32e1
   ```
5. Розставити у правильній послідовності дії, які виконуватимуться під час обчислення значення виразу. Чому дорівнюватиме значення цього виразу? Якого типу буде це значення? (Увага! У переліку можливих дій є зайві пункти).
   Вираз:
   ```
   1. - 200 % int(round(2+3**2/7, 1))
   ```
   Дії:
   1) ділення 2 на 7;
   2) піднесення 3 до квадрата;
   3) додавання 3 до 2;
   4) віднімання 200 від 1;
   5) додавання двійки до значення, отриманого на попередньому кроці;
   6) віднімання від одиниці значення виразу з попереднього кроку;
   7) ділення значення, отриманого на попередньому кроці,

на 7;

8) заокруглення отриманого на попередньому кроці значення до одного значення після коми;

9) піднесення до квадрата значення, отриманого на попередньому кроці;

10) обчислення остачі від ділення числа 200 на значення виразу з попереднього кроку;

11) відтинання дробової частини отриманого на попередньому кроці значення.

6. Виконати в інтерактивному режимі такі інструкції:

```
>>> a = 2.0
>>> b = 2.5
>>> a.is_integer()
>>> b.is_integer()
```

Чому, на Вашу думку, передостання інструкція повернула значення `True`, а остання `False`? Адже і 2.0, і 2.5 належать до типу `float`.

Якщо відповісти на запитання складно, то варото прочитати про метод `is_integer()` у документації Python або знайти інформацію про цей метод у відкритих онлайнових джерелах.

РОЗДІЛ 4
СТАНДАРТНА БІБЛІОТЕКА. МОДУЛІ ТА ЇХНЄ ВИКОРИСТАННЯ

4.1. Поняття модуля у Python

Реальні програми зазвичай не записані в одному файлі. Вони складаються з багатьох пов'язаних між собою модулів – файлів з розширенням .py, іменування яких підлягає правилам іменування змінних. З іншого боку, модуль є простором імен[23]. Імена всередині модуля називаються атрибутами[24]. Зазвичай атрибутами модулів є змінні, функції та класи.

4.1.1. Точкова нотація

Атрибут завжди пов'язаний з певним об'єктом (у нашому випадку модулем). Цей зв'язок записується у точковій нотації, синтаксис якої:

```
<ім'я_об'єкта(модуля)>.<ім'я_атрибута>
```

Наприклад, якщо файл todo.py містить змінні з запланованими завданнями:

```
task_1 = 'Прочитати книжку'
task_2 = 'Посадити дерево'
...
```

то наступний фрагмент коду, який записаний в іншому файлі, виведе текст другого завдання (значення змінної `task_2`):

```
print(todo.task_2)
```

23 Простір імен містить усі імена об'єктів, яким присвоюється значення у модулі, крім імен всередині інструкцій def і class.

24 Атрибути мають не тільки модулі, а будь-які інші об'єкти.

Насправді, якщо запустити цей фрагмент, то з'явиться повідомлення про помилку:

```
NameError: name 'todo' is not defined
```

Річ у тім, що інтерпретатору під час виконання інструкції нічого не відомо про модуль todo (файл `todo.py`), а отже, і про його атрибути. Модуль перед використанням треба імпортувати. Для цього призначена інструкція `import`, яку зазвичай пишуть на початку коду:

```
import todo
print(todo.task_2)                    # Посадити дерево
```

Якщо атрибутом модуля є функція, то її виклик відбувається теж за правилом точкової нотації[25]:

```
<ім'я_модуля>.<ім'я_функції>(<аргумент1>, <аргумент2>, ...)
```

Для того, щоб інструкція `import` змогла знайти модуль (у нашому випадку це файл todo.py), він має бути збережений у тій самій папці, що й файл основної програми, яка імпортує цей модуль. Якщо модуля там немає, то інтерпретатор намагатиметься знайти його у папках стандартної бібліотеки[26] та папках сховища сторонніх пакетів. Якщо модуль відсутній і там, то інтерпретатор видасть помилку імпорту:

```
ImportError: No module named 'todo'
```

Якщо з певних причин потрібний модуль має міститися в якомусь іншому, відмінному від зазначених вище, місці, то можна:
1) папку з цим модулем додати до сховища сторонніх пакетів;
2) зазначити шлях до цієї папки у файлі з розширенням .pth;
3) налаштувати значення змінної середовища PYTHONPATH.[27]

---

25 Використання функцій як атрибутів модулів стандартної бібліотеки Python описане у розділі 5.

26 Саме через це модулі стандартної бібліотеки, розглянутої у розділі 5, імпортуються без проблем.

27 Ці операції не розглядаються у цьому посібнику.

4.1.2. Способи імпортування модулів

Описаний вище спосіб імпорту модуля, синтаксис якого

```
import <ім’я_модуля>
```

не єдиний. Є інші способи імпорту, яким відповідають інші варіанти синтаксису доступу до атрибутів.

Можна уникнути точкової нотації, використавши синтаксис

```
from <ім’я_модуля> import *
<ім’я_атрибута>              # доступ без точкової нотації
```

Цей спосіб імпортує усі доступні в модулі ресурси. Якщо використовуються лише деякі, то їх можна імпортувати окремо:

```
from <ім’я_модуля> import <атрибут1>, <атрибут2>
<атрибут1>                   # доступ без точкової нотації
<атрибут2>()                 # доступ без точкової нотації
```

Крім того, модулю або його атрибуту можна присвоювати нові імена, використовуючи as:

```
import <ім’я_модуля> as <нове_ім’я_модуля>
# доступ з використанням точкової нотації
<нове_ім’я_модуля>.<ім’я_атрибута>


from <ім’я_модуля> import <ім’я_атрибута> as <нове_ім’я>
<нове_ім’я>          # доступ без точкової нотації
```

Вибір способу імпорту залежить від конкретної ситуації. Наприклад, якщо різні модулі містять однойменні атрибути, то для уникнення неоднозначності у коді

```
from mod_1 import f   # імпорт функції f() з модуля mod_1
from mod_2 import f   # імпорт однойменної функції з mod_2
f()                   # буде викликана функція модуля mod_2
```

доцільніше використовувати один з таких варіантів:

```
# варіант 1
from mod_1 import f as f1
from mod_2 import f as f2
f1()                          # виклик функції з першого модуля
f2()                          # виклик функції з другого модуля

# варіант 2
import mod_1, mod_2
mod_1.f()                     # виклик функції з першого модуля
mod_2.f()                     # виклик функції з другого модуля
```

4.2. Стандартна бібліотека Python

Стандартна бібліотека Python містить десятки модулів з готовим якісним кодом для роботи з різноманітними об'єктами та розширює функціонал мови. У цьому розділі наведено приклади використання кількох з них[28]. Для закріплення знань з імпорту модулів у наведених прикладах використовують різні способи імпорту.

Отримати перелік доступних атрибутів модуля можна, ввівши в інтерактивному сеансі послідовність команд[29]

```
>>> import <ім'я_модуля>
>>> dir(<ім'я_модуля>)        # список імен атрибутів
>>> help(<ім'я_модуля>)       # повна документація про модуль
```

або скористатися доступною у відкритих джерелах документацією.

---

28 У програмах можна використовувати модулі не тільки стандартної бібліотеки – співтовариство Python підтримує величезну колекцію сторонніх модулів, які потребують додаткової інсталяції. Приклад встановлення сторонньої бібліотеки matplotlib наведений у підпункті 11.1.2.).

29 Функції dir() і help() можна використовувати не тільки з модулями, а й з будь-якими іншими об'єктами.

4.2.1. Модуль math

Математичний модуль `math` містить математичні константи та функції для виконання основних арифметичних, тригонометричних, логарифмічних та інших операцій, а саме[30]:

```
acos(x)        # повертає арккосинус числа x (в радіанах)
acosh(x)       # гіперболічний арккосинус числа x
asin(x)        # арксинус числа x (в радіанах)
asinh(x)       # гіперболічний арксинус числа x
atan(x)        # арктангенс числа x (в радіанах)
atan2(y,x)     # арктангенс y/x (в радіанах) враховуючи чверть
atanh(x)       # гіперболічний арктангенс числа x
ceil(x)        # заокруглює x до найближчого більшого цілого
copysign(x,y)
# повертає число |x|, а знак збігається зі знаком числа y
cos(x)         # повертає косинус числа x (в радіанах)
cosh(x)        # гіперболічний косинус числа x
degrees(x)     # перетворює кут x з радіанної міри в градусну
e              # 2.718281828459045
erf(x)         # функція помилок
erfc(x)        # 1-erf(x)
exp(x)         # e^x
expm1(x)       # e^x - 1 (з кращою точністю, ніж exp(x)-1)
fabs(x)        # модуль (абсолютне значення) числа x
factorial(x)   # x! (факторіал числа x)
floor(x)       # заокруглює x до найближчого меншого цілого
fmod(x,y)      # остача від ділення x на y
fsum(seq)      # сума послідовності seq як дійсне число
gamma(x)       # значення гамма-функції в точці x
gcd(x,y)       # найбільший спільний дільник чисел x і y
hypot(x,y)     # гіпотенуза прямокутного трикутника з катетами x, y
```

30 Тут наведено майже повний перелік атрибутів модуля. Для наступних модулів наводитимуться лише вибрані атрибути.

```
inf           # нескінченність
isfinite(x)   # перевіряє, чи число x є скінченним
isinf(x)      # перевіряє, чи число x є нескінченним
isnan(x)      # повертає True, якщо x не є числом і навпаки
ldexp(x,i)    # повертає x * (2**i)
log(x)        # натуральний логарифм числа x
log(x,y)      # логарифм за основою y числа x
log1p(x)      # натуральний логарифм числа 1+x
log2(x)       # логарифм за основою 2 числа x
modf(x)       # повертає дробову і цілу частину числа x
pi            # 3.141592653589793
tau           # 6.283185307179586
pow(x,y)      # повертає x**y
radians(x)    # перетворює кут x з градусної міри в радіанну
sin(x)        # синус числа x (в радіанах)
sinh(x)       # гіперболічний синус числа x
sqrt(x)       # квадратний корінь числа x
tan(x)        # тангенс числа x (в радіанах)
tanh(x)       # гіперболічний тангенс числа x
trunk(x)      # відтинає дробову частину числа x (ціле число)
```

Розглянемо використання окремих функцій модуля `math` на прикладі програми для визначення кута між векторами з координатами (x_1, y_1) і (x_2, y_2):

```
import math
x1 = float(input('x1 = ')); y1 = float(input('y1 = '))
x2 = float(input('x2 = ')); y2 = float(input('y2 = '))
m1 = math.hypot(x1, y1)      # модуль (довжина) першого вектора
m2 = math.hypot(x2, y2)      # модуль (довжина) другого вектора
sp = x1*x2 + y1*y2           # скалярний добуток векторів
rad = math.acos(sp/m1/m2)    # кут у радіанах
gra = math.degrees(rad)      # кут у градусах
print('Кут між векторами у радіанах:', rad)
print('Кут між векторами у градусах:', gra)
```

4.2.2. Модуль random

Модуль `random` містить набір функцій для генерування випадкових елементів, зокрема:

```
randint(x, y)       # випадкове ціле число з проміжку [x; y]
random()            # випадкове дійсне число з проміжку [0; 1]
shuffle(L)          # перемішує елементи списку L[31]
uniform(x, y)       # випадкове дійсне число з проміжку [x, y]
```

Нижче наведено код програми, яка генерує випадкове ціле число від 1 до 100 і пропонує користувачеві відгадати його. Результатом виконання програми є число, яке вказує, на скільки одиниць користувач помилився (0 означає, що користувач вгадав число):

```
from random import *
number = randint(1, 100)
print('Відгадайте згенероване число (від 1 до 100)')
guess = int(input('Введіть Ваш варіант: '))
print('Близькість Вашої здогадки:', abs(number - guess))
```

4.2.3. Модуль fractions

Модуль `fractions` пропонує тип `Fraction` для роботи з раціональними числами. Продемонструємо деякі можливості цього модуля на такому прикладі:

```
# імпортуємо з модуля fractions тип Fraction і надаємо йому
# коротке ім’я F
from fractions import Fraction as F
x = F(2, 7)           # в x записуємо дріб (об’єкт Fraction) 2/7
y = F(14, 4)          # записуємо у змінну y дріб 14/4
print(x, y)           # 2/7 7/2 (другий дріб скорочується)
print(x-y, x*y)       # -45/14 1 (дії над дробами)
print(F('0.75'))      # 3/4 (створення дробу з рядка)
```

31 Списки описані в підрозділі 7.1.

Використання дробів дає змогу обійти обмеження точності, властиві дійсним числам, але у цьому випадку втрачається продуктивність (програма виконуватиметься довше):

```
print(3*0.1 – 0.3)    # 5.551115123125783e-17 (майже, але не 0)
print(3*F(1,10) – F(3,10))          # 0
```

4.2.4. Модуль time

Модуль `time` використовується для роботи з часом.

Розглянемо однойменну функцію `time()` модуля `time`, яка повертає кількість секунд, яка пройшла від 0 годин 0 хвилин 1 січня 1970 року (в середовищі програмістів цей момент часу називається “початок Епохи”) до системного часу комп’ютера в момент її виклику. Часто її використовують для отримання часу виконання програми. Визначимо, наприклад, скільки часу потрібно на генерування одного мільярда випадкових дійсних чисел від 0 до 1:

```
from time import time
from random import random as rnd
start = time()               # фіксація початку виконання
for i in range(1000000000):  # повторення дії мільярд разів
    rnd()                    # дія, яка повторюється
finish = time()              # фіксація часу завершення дії
print(round(finish – start, 1))    # 106,6
```

У програмі двічі викликається функція `time()`, перед початком і після завершення генерування чисел. Відповідні цим викликам значення зберігаються у змінні `start` і `finish`. Отже, різниця між цими значеннями буде дорівнювати часу виконання програми у секундах (результат заокруглено до одного знака після коми). В нашому випадку генерування тривало приблизно 1 хв 47 с.

---

32 Залежно від потужностей комп’ютера та багатьох інших чинників, час виконання програми може відрізнятися від наведеного.

4.2.5. Інші модулі

Коротка характеристика низки інших модулів стандартної бібліотеки:

`collections` – пропонує структури даних, на додачу до вбудованих списків, кортежів, словників і множин;

`copy` – надає розширені можливості для створення копій структур;

`datetime` – містить функції для роботи з датами і часом;

`itertools` – набір інструментів для реалізації ітерацій;

`multiprocessing` – дає змогу запускати додаткові процеси та розподіляти обчислення між процесорами;

`pickle` – дає змогу зберігати у файл і зчитувати з файлу об'єкти Python;

`re` – забезпечує роботу з регулярними виразами (regular expressions);

`statistics` – містить функції для статистичного аналізу даних;

`tkinter` – надає інструментарій для створення графічних інтерфейсів;

`turtle` – надає графічний інструментарій для створення простих двовимірних рисунків;

`os` – містить функції для роботи з файловою системою: зміна поточної директорії, отримання списку файлів і папок за зазначеним шляхом, створення та вилучення папок, перейменування папок і файлів, перевірка існування файлу чи папки, отримання розміру файлу та ін;

`pprint` – дає змогу відображати на екрані структури даних у зручному для сприйняття вигляді;

`setuptools` – використовується для додавання модулів до сховища сторонніх пакетів;

`sys` – інструментарій для системного програмування;

`threading` – пропонує високорівневий інтерфейс для реалізації багатопотоковості в операційній системі;

`unittest` – пропонує інструменти для автоматизованого тестування (перевірки) коду;

`urllib` – модуль для роботи з HTTP.

Питання та завдання для самоперевірки

1. Як створити модуль?
2. У модулі (файлі data.py) міститься інструкція `number = 50`. У файлі основної програми main.py міститься єдина інструкція `number = 100`. Дописати, використавши принаймі два способи імпорту, код основної програми так, щоб виводився добуток значень змінних `number` з основної програми і модуля.
3. Написати інструкцію, яка імпортує тільки одну функцію `sleep()` з модуля `time`. Як можна дізнатися, для чого призначена ця функція?
4. Обчислити усно значення виразу (інструкцію `import math` вважати виконаною):
   ```
   int(math.log(math.exp(math.degrees(math.pi))))
   ```
   Переконатись у цьому, виконавши код.
5. Чому в результаті виконання коду
   ```
   e = 3
   from math import *
   print(e-1)
   ```
   замість `2` виводиться `1.718281828459045`?
6. Записати еквівалент наведеного коду, використавши іншу функцію модуля
   ```
   import random as r
   r.random()*100
   ```

РОЗДІЛ 5
ОСНОВНІ КОНСТРУКЦІЇ МОВИ ТА ЇХНЯ РЕАЛІЗАЦІЯ

Усі попередні фрагменти коду та програми містили інструкції (команди), які виконувалися у тому ж порядку, в якому були записані. Проте часто трапляються випадки, коли подальші дії залежать від виконання (чи невиконання) деякої умови. Щоб зрозуміти, як інтерпретатор Python перевіряє умови, ознайомимося з логічним типом, логічними операторами та логічними виразами.

5.1. Логічні оператори та логічні вирази

Умови, які перевіряють під час виконання програми, записують у формі логічних виразів, результатом яких є одне зі значень логічного типу. Залежно від цього значення інтерпретатор виконує ту чи іншу послідовність інструкцій.

Прості логічні вирази зазвичай утворюються з використанням операторів порівняння: == (“дорівнює”), != (“не дорівнює”), < (“менше”), > (“більше”), <= (“менше або дорівнює”), >= (“більше або дорівнює”). Наприклад, (після логічного виразу зазначено його значення):

```
3 == 5                      # False
3 != 3.0                    # False
3 < 5                       # True
5 >= 5                      # True
```

Порівняння дійсних чисел, через їхнє наближене подання у пам’яті комп’ютера, може видати неочікувані результати:

```
print(0.1**3 == 0.001)      # False
```

Виправити цю ситуацію можна, встановлюючи близькість між числами з потрібною точністю:

```
x = 0.1**3
```

```
y = 0.001
print(abs(x-y) < 0.000001)          # True
```

Порівнювати можна не тільки числа, а й рядки, списки, кортежі та множини. Наприклад, під час порівняння рядків Python конвертує символи у їхні коди, а потім порівнює їх зліва направо.

Порівняння значень різних типів у загальному випадку не допускається – спроба порівняти число і рядок спричинить помилку:

```
5 > '2'
#TypeError:'>'not supported between instances of 'int' and 'str'
```

Із операторів порівняння можна утворювати ланцюжки:

```
1 != 2 <= 2+1 > 1.5                 # True
'A' < 'B' > 'C'                     # False
```

Для об'єднання результатів перевірок використовують логічні оператори `and` ("І"), `or` ("АБО") і `not` (логічне заперечення):

```
not (2 > 3)                         # True
2 > 3 or 3 < 4                      # True
2 + 2 == 4 and 2 * 2 != 4           # False
```

Логічні вирази можуть використовувати оператор перевірки на входження `in`, який використовується для перевірки належності об'єкта певному складеному типу:

```
'u' in 'Ukraine'                    # False
'U' in 'Ukraine'                    # True
'u' not in 'Ukraine'                # True
```

5.2. Інструкція if

Умовна інструкція `if` є складеною інструкцією (тобто, містить після символу "`:`" блоки інших інструкцій з відступами) і має такий загальний синтаксис:

```
if <логічний вираз 1>:     # перевірка умови, заданої виразом 1
```

```
    # блок виконується, якщо логічний_вираз_1 = True
    <блок інструкцій 1>
elif <логічний вираз 2>:
    # необов'язкова частина (elif може бути декілька)
    # блок виконується, якщо вираз_1 = False і вираз_2 = True
    <блок інструкцій 2>
else:
    # необов'язкова частина блок виконується, якщо всі
    # попередні логічні вирази повертають False
    <блок інструкцій 3>
```

Тобто, спочатку інтерпретатор перевіряє умову, задану логічним виразом у блоці `if`. Якщо ця умова виконується, то виконуються вкладені інструкції цього блоку. Якщо ні – то по черзі перевіряються умови усіх блоків `elif`. У цьому випадку виконаються вкладені інструкції блоку, умова в якому задовольнилася першою. Якщо жодна з умов у блоках `elif` не виконалася, то виконуються вкладені інструкції блоку `else`. Нижче наведені приклади різних варіантів інструкції `if...elif...else`.

```
# повідомлення, якщо введена літера є у слові "інформатика"
word = 'інформатика'
if input('Введіть літеру: ') in word:
    print('Введена Вами літера є у слові, word)
# виводиться інформація про парність/непарність введеного числа
number = int(input('Введіть число: '))
if number % 2 == 0:
    print('Число', number, 'парне')
else:
    print('Число', number, 'непарне')
# програма порівнює два введені числа
x = input('Введіть перше число: ')
y = input('Введіть друге число: ')
if x > y:
    print('Перше число більше за друге')
elif x < y:
```

```
    print('Перше число менше за друге')
else:
    print('Числа однакові')
```

Відступи (пробіли або символи табуляції на початку рядка) є частиною синтаксису Python. Інтерпретатор автоматично визначає межі блоків складених інструкцій за величиною відступів. Усі інструкції в межах одного блоку мають мати однаковий відступ, який має бути оформлений або пробілами, або символами табуляції (змішування не допускається)[33].

Оскільки складені інструкції можуть містити в собі інші складені інструкції, то неправильне оформлення відступів може призвести до некоректних результатів (семантичні помилки). Наприклад, синтаксично правильний код

```
age = int(input('Введіть Ваш вік: '))
if age < 18:
    print('Ви неповнолітні')
    if age < 5:
        print('Ви не школяр')
    else:
        print('Ви повнолітні')
```

видасть некоректний результат для віку 10 років: буде надруковано як `'Ви неповнолітні'`, так і `'Ви повнолітні'`. Річ у тім, що інструкція `else` стосується другого `if`, а не першого. Правильний варіант:

```
age = int(input('Введіть Ваш вік: '))
if age < 18:
    print('Ви неповнолітні')
    if age < 6:
        print('Ви не школяр')
else:
    print('Ви повнолітні')
```

33 У інтерактивному режимі, оболонка сама додає відступи після символу двокрапки, а вкладений блок завершується двома натисканнями клавіші Enter.

В окремих випадках допускається записувати умовну інструкцію в один рядок:

```
hours = int(input('Введіть кількість годин: '))
if hours > 24: print('Пройшло більше доби')
```

або

```
x = 1 if y>3 and y!=5 else 0
```

Остання інструкція є еквівалентом складеної інструкції

```
if y>3 and y!=5:
    x = 1
else:
    x = 0
```

5.3. Безоператорні логічні вирази

Логічні вирази не обов'язково мають містити оператори порівняння, логічні оператори чи оператор перевірки на входження. Будь-який об'єкт (число, рядок, функція тощо) трактується як логічний вираз, якщо він входить в умовну інструкцію[34], де відмінні від нуля числа та непорожні об'єкти завжди інтерпретуються як `True`, а нуль, порожні об'єкти та спеціальний об'єкт `None` – як `False`. Тобто, умови `if 5` та `if 'False'` (тут `'False'` є рядком, а не логічним типом) завжди виконуються, а умови `if 0` та `if ''` завжди не виконуються.

5.4. Конструкції повторення (цикли)

Більшість прикладних задач передбачає багаторазове виконання тих самих інструкцій. Логічно припустити, що існує якийсь "скорочений" синтаксис для цього. У програмуванні багатократно виконувані

34 Це саме стосується інструкції while, яка описана в підрозділі 4.6.

послідовності інструкцій називаються циклами, а кожне повторення – ітерацією.

Розрізняють два види циклів – з наперед відомою та невідомою (можуть виконуватися безкінечно або припинятися за виконання певної умови) кількістю ітерацій. Кожному з цих видів відповідає своя керуюча конструкція Python (заголовки яких починаються зарезервованими словами `for` та `while`, відповідно).

5.4.1. Функція range()

Функція `range()` як універсальний інструмент генерування цілих чисел зазвичай використовується для організації циклів з відомою кількістю ітерацій. Вона може приймати один, два або три аргументи.

У випадку з одним аргументом функція генерує цілі числа від 0 до значення аргумента, не враховуючи його:

```
range(5)              # генерує числа 0, 1, 2, 3, 4
```

Якщо функції передати два аргументи, то перший вважатиметься початком відліку:

```
range(-3, 3)          # генерує числа -3, -2, -1, 0, 1, 2
```

Третім аргументом можна задавати крок:

```
range(-10, 10, 3)     # генерує числа -10, -7, -4, -1, 2, 5, 8
```

Крок може бути від'ємним:

```
range(5, 0, -1)       # генерує числа 5, 4, 3, 2, 1
```

Якщо початкове значення менше за кінцеве, а крок від'ємний, то інтерпретатор не видасть повідомлення про помилку, просто не буде нічого згенеровано.

5.4.2. Цикл for

Цикл `for` зазвичай використовують у випадку відомої кількості повторень і має такий синтаксис:

```
for <ім'яЗмінної> in range(<аргументи>):    # заголовок циклу
    <блок інструкцій>                       # тіло циклу
```

Як і у випадку з `if`, `for` є складеною інструкцією – заголовок циклу закінчується двокрапкою, а інструкції тіла циклу оформляються з відступами. Змінна, яку використовують у заголовку циклу `for`, почергово приймає значення, згенеровані функцією `range()`. У випадку функції `range()` з одним аргументом її називають лічильником – така назва стає зрозумілою з такого коду:

```
for number in range(5):
    print(number, end = ' ')        # 0 1 2 3 4
```

Наступний фрагмент коду демонструє застосування циклу `for` для обчислення суми та добутку перших n натуральних чисел:

```
n = int(input('Введіть n: ')
suma = 0                    # початкове значення суми
dobutok = 1                 # початкове значення добутку
for i in range(1,n+1):      # надаємо змінній i значення 1...n
    suma = suma + i
    # додаємо поточне значення i до суми
    dobutok = dobutok * i
    # множимо добуток на поточне значення змінної i
print(suma,dobutok)
# наприклад, при n=100: 5050 9332...(ще 144 цифри)
```

Початкові значення суми (`0`) і добутку (`1`) перед циклом обов'язкові – інакше інтерпретатор під час спроби виконати дії у тілі циклу на першій ітерації видасть помилку `NameError: name 'suma' is not defined` (змінна циклу `i` дорівнюватиме 1, а змінні `suma` і `dobutok` на той момент не існуватимуть).

У циклах часто використовують так звані комбіновані присвоєння (у випадку додавання/віднімання їх ще називають операторами інкременту/декременту):

```
x += a                  # аналог x = x + a
```

```
x -= a                  # аналог x = x - a
x *= a                  # аналог x = x * a
x /= a                  # аналог x = x / a
x %= a                  # аналог x = x % a
```

З їхнім використанням тіло циклу з попереднього коду може бути записане простіше:

```
for i in range(1,n+1):
    suma += i
    dobutok *= i
```

Використовуючи різні комбінації аргументів функції `range()`, можна задавати крок, з яким змінюватиметься змінна циклу, організовувати зворотну лічбу та ін.

Конструкцію `for...in` також часто застосовують для генерування списків, словників і множин, порядкового обходу файлів, посимвольного обходу рядків та поелементного обходу колекцій[35].

5.4.3. Цикл while

На відміну від циклу `for`, тіло якого виконується заздалегідь відому кількість разів, інструкція `while` дає змогу організувати цикл, кількість ітерацій якого непередбачувана. Найпростіший синтаксис цієї складеної інструкції такий:

```
while <логічний_вираз>:
    <блок іструкцій>  # виконується, поки вираз = True
```

Якщо формулювання задачі потребує завершення циклу (насправді існують і реалізації нескінченного циклу, про що буде сказано нижче), то зрозуміло, що принаймні одна з інструкцій вкладеного блоку має змінювати значення деякої змінної, яка входить у `логічний_вираз` і може надати йому значення `False`.

35 Синтаксис та особливості цих операцій описані у відповідних підрозділах розділів 6-7.

Наступний фрагмент коду демонструє застосування циклу `while` для генерування випадкових цифр, допоки не буде згенеровано `0`:

```
from random import randint
number = randint(0,9)
# допоміжна інструкція (генерує першу цифру)
while number != 0:
# можна використати скорочений запис while number:
    print(number)            # друкуємо поточну цифру
    number = randint(0,9)    # генеруємо наступну цифру
```

Допоміжна інструкція потрібна для того, щоб увійти у цикл, інакше під час виконання наступного рядка виникне помилка, бо інтерпретатор не зможе обчислити значення логічного виразу `number != 0`, тому що нічого не знатиме про змінну `number`.

5.4.4. Інструкція break

Інструкцію `break` використовують для примусового виходу з циклу. Тобто, її виконання змушує інтерпретатор перейти до першої інструкції після циклу. Попередній код з використанням інструкції `break` може бути переписаний у вигляді, який не потребує допоміжної інструкції перед циклом, але містить перевірку умови в тілі циклу:

```
from random import randint
while True:                          # можна while 1:
    number = randint(0,9)            # генеруємо цифру
    if not number:    # те саме, що if number == 0:
        break
        # якщо так, то припиняємо генерування чисел
    print(number)     # інакше друкуємо поточну цифру
```

Крім того, в наведеному коді використано так званий нескінченний цикл: оскільки логічний вираз після інструкції `while` завжди дорівнює `True`, то тіло циклу виконувалося б нескінченно, якби не було інструкції `break`.

5.4.5. Інструкція continue

Інструкція `continue`, як і `break`, використовується тільки всередині циклу. Вона змушує інтерпретатор пропустити усі інструкції тіла циклу, що йдуть після неї і повернутись у заголовок циклу, тобто виконати примусовий перехід до наступної ітерації (якщо, звісно, умова у заголовку циклу виконається).

Наступний фрагмент коду демонструє застосування інструкції `continue` для генерування випадкових парних цифр, допоки не буде згенеровано 0:

```
from random import randint
number = 1     # будь-яка цифра для того, щоб увійти у цикл
while number:                # поки цифра не ноль
    number = randint(0,9)    # генеруємо цифру
    if number % 2 != 0:      # якщо цифра непарна, то
        continue             # повертаємося до заголовку циклу
    print(number)
```

5.4.6. Блок else

Цикли мають і розширений синтаксис, який містить необов'язковий блок `else`, який виконується тільки у випадку непримусового завершення циклу, тобто, коли логічний вираз у заголовку циклу while повернув значення `False` або лічильник циклу `for` перебрав усі значення. Це означає, що блок `else` є зміст використовувати лише у випадках, коли тіло циклу містить інструкцію `break`, інакше він виконуватиметься завжди.

Розглянемо іструкцію `while/else` на прикладі програми для перевірки, чи введене ціле число є простим[36]:

```
x = int(input('Введіть ціле число: '))
i = 2                # перший можливий дільник
while i <= x//2:     # дільник не перевищує половини числа
    if x % i == 0:   # якщо i є дільником, то
```

36 У цій програмі числа 0 та 1 вважають простими.

```
        print('Число не є простим')
        # виводимо на екран, що число не просте
        break
        # і забезпечуємо вихід з циклу в обхід блоку else
    i += 1     # беремо наступний можливий дільник, додавши 1
else:          # якщо інструкція break не виконається,
    print('Число просте')    # то число просте
```

Це саме можна реалізувати і з використанням циклу for:

```
x = int(input('Введіть ціле число: '))
for i in range(2, x//2 + 1):
    if x % i == 0:
        print('Число не є простим')
        break
else:
    print('Число просте')
```

5.5. Вкладені цикли

У тіло циклу можуть вкладатися інші цикли. Розглянемо це на прикладі програми, яка перевіряє, чи є введене натуральне число сумою квадратів двох інших натуральних чисел. Ідея полягає в переборі двох всіляких можливих пар натуральних чисел від 1 до заокругленого квадратного кореня з введеного числа (тому що жодне з двох шуканих чисел не можуть перевищувати цієї межі):

```
n = int(input('Введіть натуральне число: '))
limit = round(n**0.5) + 1
# limit - максимальне значення чисел у розкладі n
label = 0      # мітка, яка вказує на негативну відповідь
for n1 in range(1, limit):       # зовнішній цикл
    for n2 in range(1, limit):   # внутрішній (вкладений) цикл
        if n1**2 + n2**2 == n:
            label = 1
```

```
            break     # виходимо з внутрішнього циклу
    if label:         # якщо мітка позитивна,
        break         # то виходимо і з зовнішнього циклу
if label:             # якщо мітка позитивна,
    print(f'{n} = {n1}^2 + {n2}^2') # друкуємо розклад числа
else:                 # інакше друкуємо негативну відповідь
    print(f'{n} не є сумою квадратів двох натуральних чисел')
```

Питання та завдання для самоперевірки

1. Написати логічний вираз, який повертатиме значення, протилежне до значення виразу `a != 3 and b <= 5`.
2. Знайти синтаксичну помилку в фрагменті коду
```
if greeting = 'Слава Україні!':
answer = 'Героям слава!'
```
3. Розставити відступи у коді
```
n = input('Введіть ціле число')
if n:
if n > 0:
print('Ви ввели додатнє число')
else:
print('Ви ввели від'ємне число')
else:
print('Введене вами число – нуль')
```
4. Знайти синтаксичну помилку в коді.
```
for lambda in range(2**2):
    print(lambda)
```
5. Яке значення матиме змінна v після виконання коду?
```
v = 3
while v > 0:
    v *= 2
    break
```

6. Скільки ітерацій циклу буде виконано? Що означає символ підкреслення?

```
for _ in range(0, 10, 2):
    print(_)
```

7. Які з наведених п’яти циклів ніколи не припиняться? Чому? Що буде надруковано у циклах, виконання яких завершиться після скінченної кількості ітерацій.

```
# перший цикл
for i in range(10, 1, -2):
    i = 10
    print(i, end='')
# другий цикл
x = 10
while 1:
    if not x:
        break
    x -= 1
    print(x, end='')
# третій цикл
x = 10
while 1:
    if not x:
        break
    x -= 3
    print(x, end='')
# четвертий цикл
x = 10
while x//2 !=1:
    if x>0:
        print(x, end='')
        continue
    break
```

```
# п’ятий цикл
i = x = 3
while i != 0 or x > 1:
    print(x**i, end='')
    x = x-1
```

8. Переписати код, замінивши цикл `while` на цикл `for`.

```
a = float(input())
d = float(input())
n = int(input(()))
i = 2
while i <= n:
    a += d
    i += 1
print(a)
```

Що виконує цей код? Яким одним виразом можна замінити цикли?

РОЗДІЛ 6
РОБОТА З РЯДКАМИ ТА ФАЙЛАМИ

6.1. Рядки

6.1.1. Літерали рядків

Послідовність будь-яких символів, записана в одинарних або подвійних лапках, називається рядком (тип `str`). Літерали `''` (або `""`) презентують порожній рядок. Подвійні лапки зазвичай використовують, якщо серед символів, що складають рядок, є апостроф, наприклад, `"П'ять – ціле число"` (і навпаки, одинарні лапки використовують, якщо рядок містить подвійні лапки).

Як аргумент функції `print()` можна використовувати рядок у потрійних лапках (`'''` або `"""`), записаний у кілька рядків – тоді він буде виведений на екран теж у кілька рядків:

```
print('''Речення, яке ви бачите, записане
у два рядки''')
```

Навіть якщо символи рядка утворюють число (`'100'`), дату (`'27.05.2019'`), IP-адресу (`'192.168.0.1'`) тощо, то для інтерпретатора все це звичайний текст (послідовність символів) без жодного змісту.

6.1.2. Символи

Мова Python, на відміну від багатьох інших мов програмування, не має окремого символьного типу – для неї кожен поодинокий символ належить до типу `str`, тобто є рядком, який складається з одного символу. Кожному символу поставлено у відповідність унікальне ціле число (код символу). Для перетворення символу в код і навпаки призначені вбудовані функції `ord()` і `chr()`:

```
code_A = ord('A')
code_a = ord("a")
```

```
char1041 = chr(1041)
print(code_A, code_a, char1041)             # 1040   1072   Б
```

Із виводу функції print() випливає, що: 1) великі та малі літери – це різні символи; 2) сусіднім літерам відповідають сусідні коди (цілі числа).

6.1.3. Керуючі символи

Якщо ввести розглянутий вище літерал рядка в інтерактивному сеансі, то отримаємо посимвольний запис цього рядка:

```
>>>'''Речення, яке ви бачите, записане
у два рядки'''
'Речення, яке ви бачите, записане\nу два рядки'
>>>
```

У виводі з'явився спеціальний символ `'\n'`, який інтерпретується як кінець рядка. Тобто, замість

```
print('''Речення, яке ви бачите, записане
у два рядки''')
```

для отримання такого ж результату можна написати

```
print('Речення, яке ви бачите, записане\nу два рядки')
```

або, пригадавши, що кожне виведення функції `print()` завершується символом нового рядка:

```
print('Речення, яке ви бачите, записане')
print('у два рядки')
```

Інший часто вживаний символ – символ горизонтальної табуляції `'\t'`:

```
print('1\t2\t3\t...')              # 1    2    3    ...
```

Керуючі символи починаються з оберненої косої риски \, за якою йдуть один чи кілька інших символів (уся послідовність символів трактується інтерпретатором як один символ). Обернена коса риска означає те, що наступний за нею символ (символи) мають трактуватися

по особливому. Її часто використовують для екранування спеціальних символів (\\, \", \' та інші), які без неї трактувалися би по-іншому, наприклад,

```
print('П\'ять – ціле число')          # П'ять – ціле число
print('one\\two\\three')              # one\two\three
```

Відсутність оберненої косої риски в першому випадку призвела би до появи синтаксичної помилки, у другому було би надруковано

```
one     wo      hree
```

Щоб уникнути екранування символів, в Python використовують неформатовані рядки, які починаються символом `r`:

```
print(r'one\two\three')               # one\two\three
```

6.1.4. Доступ до елементів рядка. Індекси та зрізи

Кожен елемент рядка (символ) має індекс – унікальне ціле число, яке однозначно вказує на порядковий номер символу у рядку. Індекси можна відраховувати з початку рядка (тоді нумерація починається з нуля: 0, 1, 2, ...), а можна з кінця (тоді індекси від'ємні: -1, -2, -3 і т. д.):

```
s = 'рядок'
print(s[0], s[2], s[-3])              # р д д
```

Але варто пам'ятати, що рядок – незмінюваний тип даних, тобто намагання змінити, вставити або вилучити елемент рядка спричинить помилку:

```
s[0] = 'P'
# TypeError: 'str' object does not support item assignment
```

Крім того, помилка виникне і у випадку, коли елемента з вказаним індексом не існує:

```
print(s[5])    # IndexError: string index out of range
```

Операція зрізу `[i:j]` повертає частину рядка, починаючи з символу з індексом `i` та закінчуючи символом з індексом `j-1` (подібно до того, як генерувалася послідовність функцією `range()`):

```
s = 'рядок'
print(s[1:3])         # яд
```

У цьому випадку наявність індексів не обов'язкова:

```
print(s[:3])
# ряд (відсутність першого індекса означає "від початку")
print(s[1:])
# ядок (відсутність другого індекса означає "до кінця")
print(s[:])
# рядок ("від початку до кінця" – копія рядка)
```

Повний синтаксис зрізу містить три індекси: `[i:j:k]`. Третім індексом `k` задається крок, який за його відсутності приймається таким, що дорівнює 1:

```
print(s[1::2])
# яо (від елемента з індексом 1 до кінця з кроком 2)
```

У зрізах допускається використання від'ємних індексів, де від'ємний крок вказує на вибір елементів справа наліво:

```
print(s[1:-2])
# яд (передостанній символ (з індексом -2) вже не включається)
print(s[3:0:-1])
# одя (перший елемент (з індексом 0) не включається)
print(s[-3:])
# док (три останні елементи)
```

Зрізи дають змогу змінювати рядки (насправді рядок не змінюється, а новоутворений у правій частині інструкції присвоєння рядок перезаписується у змінну з тією ж назвою):

```
s = s[0].upper()+s[1:]
```

```
# функція upper() перетворює символ у верхній регістр
print(s)              # Рядок
```

6.1.5. Конкатенація (об'єднання) та повторення рядків

Рядки можна об'єднувати, утворюючи нові рядки. Така операція називається конкатенацією і виконується за допомогою оператора +:

```
name = 'Роман'
s = 'Мене звати ' + name + '.'
print(s)              # Мене звати Роман.
```

Спроба об'єднати рядок з об'єктом іншого типу спричинить помилку, тому цей об'єкт має бути попередньо перетворений в рядок. Наприклад, для перетворення числових типів `int` і `float` у тип `str` використовується однойменна функція `str()`:

```
name = 'Роман'; age = 43
s = 'Мене звати ' + name + '.'
s += ' Мені ' + str(age) + ' роки.'
print(s)       # Мене звати Роман. Мені 43 роки.
```

Для об'єднання кількох однакових рядків (повторення певної послідовності символів) використовують оператор *****:

```
s = '-'*5 + 'abc'*2 + '-'*5
print(s*3)
# -----abcabc----------abcabc----------abcabc-----
```

6.1.6. Форматування рядків

Крім конкатенації, Python пропонує й інші способи формування/форматування рядків. Історично першим є просте позиційне форматування з використанням оператора **%**, який визначає певний специфікатор формату (`%s` – рядок, `%d` – ціле число, `%f` – дійсне число та ін.).

```
name = 'Роман'
s = 'Мене звати %s.' % name
print(s)       # Мене звати Роман
```

У наведеному фрагменті коду специфікатор формату %s вказує на позицію, в яку має бути вставлене значення змінної name у вигляді рядка. Якщо у рядку має бути кілька підстановок, то перелік змінних треба подавати як кортеж[37], наприклад,

```
name = 'Роман'
age = 43
s = "Мене звати %s. Мені %d роки." % (name, age)
print(s)       # Мене звати Роман. Мені 43 роки.
```

Оператор % може приймати аргумент і у вигляді словника[38]:

```
person = {'name': 'Роман', 'age': 43}
s = "Мене звати %(name)s. Мені %(age)d роки." % person
```

Специфікатори формату після оператора % можуть містити дані про ширину поля (у символах), відведену під значення (використовується для друку у кілька стовпців з різними способами вирівнювання). Продемонструємо це на прикладі програми, яка виводить три рядки з випадковою кількістю символів "+" (від 1 до 20):

```
from random import randint
for i in range(3):
    n = randint(1,20)
    print('%-3d%21s' % (n, '+'*n))
```

Результатом виконання програми будуть три надруковані рядки, кожен з яких містить випадково згенероване число і відповідну йому кількість плюсів. На числа відведено три позиції (знак мінус вказує на

37 Для розуміння цього способу форматування варто ознайомитися з матеріалом підрозділу "Кортежі" розділу 7.

38 Для розуміння цього способу форматування варто ознайомитися з матеріалом підрозділу "Словники" розділу 7.

вирівнювання за лівим краєм), а на плюси – 21 позицію (вони будуть вирівняні за правим краєм):[39]

```
2                     ++
1                      +
11           ++++++++++
```

Якби ширини полів не зазначалися, тобто функції `print()` передавався би рядок `'%d %s' % (n, '+'*n)`, то результат був би таким:

```
2 ++
1 +
11 ++++++++++
```

Для відображення дійсних чисел після ширини поля через крапку можна зазначати кількість десяткових знаків, наприклад, число “пі” можна надрукувати з різною точністю:

```
from math import pi
print(1,'%5.1f' % pi)        # 1   3.1
print(2,'%5.2f' % pi)        # 2  3.14
```

У цьому випадку позиція, яку займає символ крапки “.” враховується у ширині поля. Ширина поля не є обов’язковою – можна задати лише кількість десяткових знаків:

```
print(1,'%.1f' % pi)         # 1 3.1
print(2,'%.2f' % pi)         # 2 3.14
```

У третій версії Python з’явилася можливість форматування з використанням функції `format()`[40]. У наведеному нижче коді цифрами у фігурних дужках позначені порядкові номери аргументів функції (за потреби після двокрапки передається специфікатор формату):

```
s = 'Мене звати {}.'.format(name)
s = "Мене звати {0}. Мені {1:d} роки.".format(name, age)
```

39 Це зразок результату виводу, бо кожен запуск генеруватиме інші числа.

40 Пізніше була перенесена і в Python 2.7.

```
s = "Мене звати {my_name}. Мені {my_age} роки."\
    .format(my_name=name, my_age=age)
```

Цей спосіб також підтримує параметри форматування, які зазначають після символу ":". Тип вирівнювання задається символами "<" (за лівим краєм), ">" (за правим краєм), "^" (по центру), після яких зазначається ширина поля:

```
from random import randint
for i in range(3):
    n = randint(1,20)
    print('{0:<3}{1:>21}'.format(n, '+'*n))
```

Параметри форматування дійсних чисел такі ж самі, як і для оператора %, наприклад,

```
import math
print('{0:.3f}'.format(math.pi))            # 3.142
```

Починаючи з версії Python 3.6, доступні форматовані літерали рядків (f-рядки), які дають змогу підставляти не тільки значення змінних, а й значення виразів. Інтерпретатор Python розпізнає f-рядок за символом f перед відкриваючими лапками:

```
name, now_year, year_of_birth = 'Роман', 2019, 1976
s = f"Мене звати {name}. Мені {now_year - year_of_birth} роки."
```

Є принаймні ще один спосіб форматування – шаблонні рядки. Оскільки вони містяться у модулі string стандартної бібліотеки, то особливості їхнього використання тут не розглядаються.

6.2. Функції для роботи з рядками та методи рядків

Як і інші типи, рядки підтримують низку функцій та методів, зокрема:

```
s = 'Це рядок'
len(s)
```

```
# к-сть символів у рядку (з пробілами та керуючими символами)
s.upper()
# переводить символи у верхній регістр ('ЦЕ РЯДОК')
s.lower()
# переводить символи у нижній регістр ('це рядок')
s.capitalize()
# перший символ у верхній регістр ('Це рядок')
s.replace('ок','ки')
# заміна усіх входжень 'ок' на 'ки' ('це рядки')
s.find('ряд')
# повертає індекс першого входження 'ряд' (3)
s.count('ряд')
# повертає кількість входжень 'ряд' (1)
s.isalpha()
# перевіряє, чи рядок містить тільки літери (False)
s.isdigit()
# перевіряє, чи рядок містить тільки цифри (False)
s.startswith('Це')
# перевіряє, чи рядок починається з 'Це' (True)
s.endswith('.')
# перевіряє, чи рядок закінчується крапкою (False)
s.strip()
# вилучає пробіли на початку та в кінці рядка
```

Важливо! Оскільки рядки є незмінюваними типами, то методи рядків мають бути частиною інструкцій присвоєння. У наступному коді рядок s не зміниться (інтерпретатор просто створить новий об'єкт рядка, але не прив'яже його до змінної):

```
s = 'Це рядок'
s.replace('ок','ки')
print(s)                                    # Це рядок
```

Щоб усе спрацювало, код має бути таким:

```
s = 'Це рядок'
```

```
s = s.replace('ок','ки')
print(s)                                    # Це рядки
```

Але:

```
s = 'Це рядок'
print(s.replace('ок','ки'))                 # Це рядки
print(s)                                    # Це рядок
```

Окремо виділимо методи перетворення рядків у списки, і навпаки. Метод `split()` повертає об'єкт іншого типу – список підрядків. За замовчуванням рядок розбивається на підрядки за символом пробілу, але аргументом методу можна задати будь-який інший розділювач, який не обов'язково складається з одного символу:

```
sentence = 'Я прочитав про програмування мовою Python'
words = sentence.split()
print(words)
# ['Я', 'прочитав', 'про', 'програмування', 'мовою', 'Python']
pro = sentence.split('про')
print(pro)
# ['Я ', 'читав ', ' ', 'грамування мовою Python']
```

Використовуючи метод `split()` та індекси, можна написати компактний код, наприклад, для отримання розширення файлу з його імені:

```
filename = 'фото_1.jpg'
print(filename.split('.')[-1])      # jpg
```

Метод `join()` утворює новий рядок шляхом конкатенації елементів списку, який передається аргументом, де рядок, до якого застосовується метод, є розділювачем:

```
c = '--'.join(words)
print(c)      # Я--прочитав--про--програмування--мовою--Python
```

6.2.1. Перевірка на входження та посимвольний обхід рядка

Часто (в умовних конструкціях) потрібно мати відповідь на питання, чи містить рядок деякий підрядок. Для цього у логічних виразах використовують оператор `in`:

```
s = 'Це рядок'
s1 = 'ряд'
print(s1 in s)        # True
```

Обхід елементів рядка (символів) відбувається з використанням інструкції `for`:

```
s = 'Це рядок'
for symb in s:
    print(symb.upper(), end = '')   # буде надруковано ЦЕ РЯДОК
```

У наведеному вище коді змінна `symb` почергово набуває значень символів рядка (`'Ц'`, `'е'`, `''`, `'р'` і т. д.), а у тілі циклу ці значення друкуються у верхньому регістрі.

Посимвольний обхід рядка може відбуватися і з використанням індексів:

```
for i in range(len(s)):
    print(s[i].upper(), end = '')
```

або

```
i = 0
while i < len(s):
    print(s[i].upper(), end = '')
    i += 1
```

6.3. Файли

Програма може приймати вхідні дані не тільки з клавіатури за допомогою функції `input()`, а й з файлів. Головні переваги цього методу

– можливість отримання великих обсягів вхідних даних, які можуть використовуватися одночасно великою кількістю програм. Це саме стосується і виводу – альтернативою виводу результатів виконання програми на екран за допомогою функції `print()` є запис результатів у текстовий файл. У цьому випадку дані, які зчитуються з файлу і записуються у файл, завжди є рядками (тип `str`).

Для роботи з файлами в Python призначена вбудована функція `open()`, яка створює об'єкт файлу на підставі переданих їй як аргументи імені файлу та режиму роботи з ним (читання, запис тощо).

6.3.1. Запис у файл

Для того, щоб створити об'єкт файлу для подальшого запису в нього, потрібно передати функції `open()` першим аргументом ім'я цього файлу, а другим – рядок `'w'`, наприклад,

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

Змінна `f`, якій присвоюється результат функції `open()`, називається дескриптором файлу. Інтерпретатор шукатиме файл *result.txt* у тій самій папці, де записано код програми. Якщо файлу з таким іменем немає, то він буде створений. Якщо файл з таким іменем є, то (увага!) усі дані в ньому будуть стерті без попередження. Тому, щоб відкрити існуючий файл в режимі "додавання в кінець", замість `'w'` другим аргументом функції `open()`, потрібно передати `'a'`.

Для запису у файл використовується метод `write()`:

```
f.write('Квадрати перших десяти натуральних чисел:\n')
for i in range(1, 11):
    f.write(str(i**2) + '\t')
```

Кожна порція даних додається в кінець файлу, але не з нового рядка. Саме тому у попередньому коді використовують символи нового рядка та табуляції. Крім того, числа перед записом у файл конвертуються в рядки функцією `str()`. Записати інформацію у файл можна і за

допомогою функції `print()`, передавши дескриптор файлу іменованому аргументу `file`. Тіло циклу з попереднього коду мало б тоді такий вигляд:

```
print(str(i**2) + '\t', file = f)
```

Завершивши запис у файл, його потрібно закрити:

```
f.close()
```

Незакриття файлу може призвести до втрати даних, метод `close()` гарантує, що дані будуть записані у файл. У підсумку вміст файлу *result.txt* буде таким:

```
  Квадрати перших десяти натуральних чисел:
1     4     9     16      25      36      49      64      81      100
```

6.3.2. Зчитування з файлу

Для відкриття файлу з метою подальшого зчитування даних з нього потрібно передати функції `open()` другим аргументом `'r'`[41]:

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

Інтерпретатор шукатиме файл *data.txt* у тій самій папці, де збережена програма. Якщо файлу з таким іменем немає, то буде ініційована помилка.

Зчитати інформацію з файлу можна різними способами.

Перший спосіб полягає в записі всього вмісту файлу в один рядок методом `read()`:

```
allText = f.read()
```

Якщо намагатися ще раз виконати попередню інструкцію, то отримаємо порожній рядок, тому для повторного зчитування даних з

41 Оскільки значення 'r' передається за замовчуванням, то у разі відкриття файлу для читання другий аргумент може бути опущений: f = open('result.txt').

файлу його треба наново відкрити методом `open()`[42] або перевстановити вказівник на початок методом `seek()`:

```
f.seek(0)
```

Другий спосіб дає змогу обробляти дані з файлу порядково, використовуючи цикл `for`:

```
for line in f:
    ...
```

Змінна `line` почергово прийматиме значення рядків файлу, тобто послідовності символів до наступного символу кінця рядка `'\n'` включно з ним. Наприклад, якщо у файлі data.txt містяться числа (по одному в кожному рядку), то їхню суму можна порахувати так:

```
f = open('data.txt', 'r')
s = 0
for line in f:
    s += float(line.strip())
f.close()
```

Метод `strip()` прибирає з кожного рядка файлу символ `'\n'`, а функція `float()` конвертує цифри, які залишилися, у числовий тип.

Для порядкового зчитування даних з файлу можна також використати метод `readline()`, помістивши його в тіло циклу `while`:

```
f = open('data.txt')
s = 0
while True:
    line = f.readline()
    if len(line) == '':
        break
    s += float(line.strip())
f.close()
```

42 У цьому випадку закривати файл методом close() не обов'язково.

Тут використано те, що у разі досягнення кінця файлу метод `readline()` повертає порожній рядок.

Третій спосіб використовує метод `readlines()`, який повертає список рядків файлу:

```
f = open('result.txt')
l = f.readlines()
print(l)
# ['Квадрати перших десяти натуральних чисел:\n',
# '1\t4\t9\t16\t25\t36\t49\t64\t81\t100\t']
f.close()
```

6.3.3. Режими відкриття файлів

Крім розглянутих вище трьох режимів відкриття файлів (для запису, для додавання в кінець і для зчитування), які задаються значеннями аргументів `'w'`, `'a'` і `'r'`, відповідно, існує ще режим створення нового файлу для запису (якщо файл із зазначеним іменем вже існує, то буде виведено повідомлення про помилку). Йому відповідає значення `'x'`.

Крім того, передбачалося, що файли містили тільки рядки з текстовими даними (ASCII або UTF-8). Під час роботи з нетекстовими файлами (зображеннями, аудіозаписами тощо), потрібно зазначати, що вони мають відкриватися у двійковому режимі. Це виконується додаванням символу `'b'` перед потрібним режимом: `'bw'`, `'ba'`, `'br'`, `'bx'`.

6.3.4. Інструкція with

Інструкція `with` є альтернативою використання функції `open()` разом з методом `close()`[43] і має такий синтаксис:

```
with open(ім'я_файлу, режим) as ім'я_дескриптора_файлу:
    # блок інструкцій
```

43 Це стосується використання with з файловими об'єктами. Насправді інструкція менеджера контексту with пропонує ширші можливості.

Вона теж створює змінну-дескриптор файлу й автоматично закриває файл після завершення виконання блоку інструкцій. Наприклад, код з підрозділу “Запис у файл” з використанням інструкції `with` виглядатиме так:

```
with open('result.txt', 'w') as f:
    f.write('Квадрати перших десяти натуральних чисел:\n')
    for i in range(1, 11):
        f.write(str(i**2) + '\t')
```

6.3.5. Повне ім’я файлу

Імена файлів `'data.txt'` і `'result.txt'`, які використовувалися вище, є короткими іменами файлів. Приймаючи їх, функція `open()` шукає файли з такими іменами у поточній папці, тобто в тій же папці, у якій збережена програма. Для встановлення зв’язку з файлами, збереженими в інших місцях файлової системи, використовують повні імена файлів.

Повне ім’я файлу містить шлях (ланцюжок папок) до зазначеного файлу. Елементи шляху відокремлюються символом `'/'` (для операційних систем Linux і macOS) або `'\'` (Windows). Щоб уникнути керуючих символів у шляху, потрібно використовувати екранування або неформатовані рядки.

Крім того, повне ім’я файлу може бути абсолютним або відносним. В абсолютному повному імені файлу шлях починається з кореневої папки (диску), у відносному – з поточної папки, яка позначається `'..'`, наприклад,

```
# приклади абсолютних шляхів:
'/home/user/code/numbers.txt'
'C:\\user\\code\\numbers.txt'       # екранування
r'C:\user\code\numbers.txt'         # неформатований рядок

# приклади відносних шляхів:
```

```
'../user/code/numbers.txt'
r'..\code\numbers.txt'
```

Питання та завдання для самоперевірки

1. Що буде виведено на екран?
   ```
   s = 'yes\no'
   print(s.upper())
   print(s)
   ```
2. Попри те, що рядок є незмінюваним типом, як можна реалізувати зміну рядка `s`, яка полягає у вилученні його третього за порядком символу (вважаємо, що рядок містить принаймні три символи)?
3. Як, використовуючи зрізи, із заданого рядка утворити новий, змінивши порядок символів на протилежний?
4. Написати код, який перетворює введену велику латинську літеру на малу, не використовуючи метод `lower()`.
5. Не використовуючи методу `count()`, написати програму для обчислення кількості символів-цифр у рядку.
6. Яке значення використовується за замовчуванням для аргументу режиму обробки файлу функції `open()`?
7. Написати програму, яка записує у файл квадрати чисел від 1 до 100 у вигляді
   ```
   1 4 9 16 25 36 49 64 81 100 121 144
   ```
8. Нехай у файлі *data.txt* міститься інформація про прибутки за місяцями року:
   ```
   Січень: 125.4
   Лютий: 1000.0
   ...
   Грудень: 200.35
   ```

Написати програму, яка обчислює загальний річний прибуток.

9. Виконання яких (з наведених нижче) інструкцій зумовить: а) синтаксичну помилку (syntax error); б) помилку виконання (runtime error). Для інструкцій, які будуть виконані без помилок, вказати значення, які будуть виведені на екран.

```
1) a = "0"
   print(a[0])
2) a = "1"
   print(a[1])
3) a = "0"
   print(a[-1])
4) a = "0"
   print(a+2)
5) a = "0"
   print(a+'2')
6) a = "0"
   print(a*2)
7) a = "0"
   print(int(a)+2)
8) a = "0"
   print(a**2)
9) a = "0'
   print(a)
10) a = "0"
    print(a(0))
11) print('import'(1))
12) print('import'[1])
```

РОЗДІЛ 7
ОСНОВНІ СТРУКТУРИ ДАНИХ PYTHON

7.1. Списки

Структура даних – це спосіб організації даних (разом з набором операцій над ними), за яким сукупність значень розглядають як одне ціле. Наприклад, рядок (тип `str`) – це структура даних, яка об'єднує послідовність символів у текст. Мова Python, крім рядків, підтримує інші структури даних, зокрема списки, кортежі, словники та множини. Усі чотири використовують для зберігання колекцій об'єктів, але мають свої унікальні особливості.

Список (тип `list`) – це впорядкована змінювана послідовність (колекція) об'єктів, які називаються елементами списку. Списки за логікою організації схожі на рядки, а як структура даних мають багато спільного з масивами в інших мовах програмування[44], але суттєво відмінні від них. По-перше, немає потреби наперед зазначати кількість елементів (довжину) списку. По-друге, елементами списку можуть бути значення будь-яких типів: числа, рядки, інші списки тощо. Крім того, тип елементів того самого списку може бути різний (елементами рядка можуть бути лише символи, тобто рядки одиничної довжини). По-третє, списки – змінюваний тип, тобто підтримують додавання, вилучення та зміну елементів.

7.1.1. Подання списків

Літералом списку в Python є послідовність об'єктів, записаних через кому у квадратних дужках, наприклад,

```
[3., 3.1, 3.14, 3.142, 3.1416]    # список наближень числа “пі”
['зелений', 'жовтий', 'червоний'] # кольори світлофора
```

44 Також не треба плутати списки Python зі зв'язаними списками (linked lists).

```
[1, ['Yes', 'No'], 2.1, 'Maybe']    # елементів різних типів
[]                                  # порожній список
```

Список може бути записаний у кілька рядків:

```
rainbow = ['red', 'orange',
           'yellow', 'green', 'blue',
           'indigo', 'violet']
```

Інтерпретатор вважатиме вираз завершеним, тільки коли знайде закриваючу квадратну дужку.

Якщо елементом списку є вираз, то інтерпретатор обчислюватиме значення цього виразу:

```
x = 10
print([x-1, x**2-1, x**3-1])        # [9, 99, 999]
```

Списки можна об'єднувати, використовуючи оператор "**+**" (операція схожа на конкатенацію рядків):

```
pi = [3., 3.1, 3.14, 3.142, 3.1416]
pi = pi + [3.14159, 3.141593]
pi += [3.1415927]
print(pi)
# [3.0, 3.1, 3.14, 3.142, 3.1416, 3.14159, 3.141593, 3.1415927]
```

Оператор * дає змогу розмножувати список однаковими значеннями:

```
print([1, 'один']*3)                # [1, 'один', 1, 'один',
1, 'один']
```

Список також може бути утворений з будь-якої послідовності елементів. Для цього потрібно передати цю послідовність функції `list()` як аргумент:

```
s = 'Це рядок.'
l = list(s)
print(l)      # ['Ц', 'е', ' ', 'р', 'я', 'д', 'о', 'к', '.']
print(list(range(10)))       # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

7.1.2. Індекси та зрізи

Кожен елемент списку має індекс – унікальне ціле число, яке однозначно визначає порядковий номер елемента у списку. Індекси можна відраховувати з початку списку (тоді нумерація починається з нуля: 0, 1, 2, ...), а можна з кінця (тоді індекси від'ємні: -1, -2, -3 і т. д.):

```
spysok = [1, ['Yes', 'No'], 2.1, 'Maybe']
print(spysok[1])                          # ['Yes', 'No']
print(spysok[0], spysok[2], spysok[-2])   # 1 2.1 2.1
```

Щоб отримати доступ до елементів вкладеного списку, індекси записують послідовно:

```
print(spysok[1][0], spysok[1][1])         # Yes No
```

Оскільки список – змінюваний тип даних, то його елементи можна змінювати:

```
spysok[1] = 'Maybe'
print(spysok)               # [1, 'Maybe', 2.1, 'Maybe']
```

Якщо елемента з зазначеним індексом не існує, то буде ініційована помилка:

```
print(spysok[4])      # IndexError: list index out of range
```

Зрізи, як і у випадку рядків, дають змогу отримати певну підмножину елементів списку. У цьому випадку сам список не змінюється – результат зрізу записується у новий список. Запис зрізу має вигляд `<ім'я_списку>[<початок>:<кінець>:<крок>]`, де жоден з параметрів у квадратних дужках не є обов'язковим. Пам'ятаємо також, що діапазон елементів задається інтервалом `[<початок>; <кінець>]`. Продемонструємо деякі види зрізів списків на прикладі:

```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue',
'indigo', 'violet']
print(rainbow [:])
# ['red','orange','yellow','green','blue','indigo','violet']
print(rainbow [:3])          # ['red', 'orange', 'yellow']
```

```
print(rainbow [2:4])          # ['yellow', 'green']
print(rainbow [1::2])         # ['orange', 'green', 'indigo']
print(rainbow [::-1])
# ['violet','indigo','blue','green','yellow','orange','red']
```

7.1.3. Функції для роботи зі списками та методи списків

Функція `len()` повертає кількість елементів списку:

```
len(rainbow)                  # 7
len([1, ['Yes', 'No'], 2.1, 'Maybe'])
# 4 (список ['Yes', 'No'] – 1 елемент)
len([])                       # 0
```

Методи списків розглянуті нижче на прикладі списку `someList`[45]:

```
someList = [1, 2, 'three']
someList.pop()
# вилучає елемент з кінця списку: [1, 2]
someList.append(3)
# додає елемент в кінець списку: [1, 2, 3]
someList.pop(1)
# вилучає і повертає елемент з індексом 1: [1, 3]
someList.insert(1, 2)
# вставляє 2 на місце з індексом 1: [1, 2, 3]
someList.extend([2, 4])
# додає в кінець елементи списку [2, 4]: [1, 2, 3, 2, 4]
someList.remove(2)
# вилучає перше входження елемента 2: [1, 3, 2, 4]
someList.sort()
# сортує елементи списку: [1, 2, 3, 4]
someList.reverse()
# розміщує елементи в зворотному порядку: [4, 3, 2, 1]
```

45 Отримати повний перелік атрибутів (у тім числі й методів) списків можна, виконавши в інтерактивному режимі команду dir([]).

```
someList.count(3)
# повертає кількість входжень елемента 3: 1
someList.index(3)
# повертає індекс елемента 3: 1
someList.clear()
# очищує список: []
```

У розділі 6 розглянуто методи `split()` та `join()`. Оскільки вони безпосередньо стосуються списків, то наведемо приклади їхнього використання:

```
s = '-'.join(['a','b','c'])
# утворює зі списку ['a','b','c'] рядок'a-b-c'
l = s.split('-')
# утворює з рядка 'a-b-c' список ['a','b','c']
```

7.1.4. Перевірка на входження та поелементний обхід списку

Часто (в умовних конструкціях) треба мати відповідь на питання, чи містить список деякий елемент. Для цього використовується оператор `in`:

```
trafficLight = ['зелений', 'жовтий', 'червоний']
color = input('Введіть колір: ')
if color in trafficLight:
    print(color, 'є серед кольорів світлофора')
else:
    print('Введеного кольору немає серед кольорів світлофора')
```

Поелементний обхід списку відбувається з використанням циклу `for`:

```
for item in my_list:
```

У цій конструкції змінна `item` по черзі набуває значення елементів списку `my_list`. Наприклад, такий цикл виводить на екран міста зі списку `cities`:

```
cities = ['Kyiv', 'London', 'New York']
for city in cities:
    print(city)
```

Якщо потрібен індекс елемента списку, то реалізується цикл за індексами:

```
for index in range(len(my_list)):
```

або використовується вбудована функція `enumerate()`:

```
for index, item in enumerate(my_list):
```

У першому випадку доступ до елемента списку реалізується виразом `my_list[index]`, у другому змінна `index` набуває значення з діапазону індексів списку, а змінна `item` відповідні цим індексам елементи.

Вивід на екран міст зі списку `cities` разом з їхніми порядковими номерами виглядатиме так (пам'ятаємо, що нумерація в Python починається з 0):

```
for i in range(len(cities)):
    print(f'{i+1}. {cities[i]}')
for i, city in enumerate(cities):
    print(f'{i+1}. {city}')
```

Як альтернативний варіант можна використати інструкцію `while`, але вона виглядає громіздкішою через ручну ініціалізацію та інкремент лічильника `i`:

```
i = 0
while i < len(cities):
    print(f'{i+1}. {cities[i]}')
    i += 1
```

Для обходу вкладених списків (або списків, елементами яких є рядки чи колекції), використовуються вкладені цикли[46].

46 Вкладені цикли описані у підрозділі 4.7.

7.1.5. Заповнення списків під час виконання програми

Розглянутий вище спосіб утворення списку, який полягає в безпосередньому заданні його елементів у квадратних дужках, називається літеральним. На практиці часто списки формуються під час виконання програми. Продемонструємо цей спосіб на прикладі програми, яка формує список унікальних символів у рядку[47]:

```
s = 'Київ – столиця України'
uniqueSym = []
# створюємо порожній список для унікальних символів
for symbol in s:      # обходимо рядок s посимвольно
    if symbol not in uniqueSym:     # якщо символа немає у
списку,
        uniqueSym.append(symbol)    # то додаємо його
print(uniqueSym)
# ['К', 'и', 'ї', 'в', ' ', '–', 'с', 'т', 'о', 'л',
# 'ц', 'я', 'У', 'к', 'р', 'а', 'н']
```

7.1.6. Генератори списків

Генератори списків призначенні для автоматизації створення списків[48], елементи яких утворюються згідно з певними правилами. Загальний синтаксис генератора списку:

```
[<вираз> for <змінна> in <колекція> if <умова>]
```

Наприклад, згенерувати список квадратів парних двоцифрових натуральних чисел можна так:

```
l = [n**2 for n in range(10,100) if n%2 == 0]
```

Цей запис еквівалентний такій реалізації:

```
l = []
```

47 Цю програму простіше реалізувати, використовуючи множини, які розглянуті в підрозділі 7.4.

48 Існують також генератори словників і множин (див. підрозділи 7.3 і 7.4).

```
for n in range(10,100):
    if n%2 == 0:
        l.append(n**2)
```

Інструкцію `if` можна опустити. Згаданий вище список можна згенерувати, замінивши перевірку остачі від ділення на 2 кроком 2:

```
l = [n**2 for n in range(10,100,2)]
```

Інколи трапляється, що потрібно щось порахувати на основі згенерованого списку, але самого списку не треба. Щоб ефективно використати оперативну пам'ять, можна скористатися вбудованим виразом-генератором, синтаксис якого відрізняється лише дужками:

```
(<вираз> for <змінна> in <колекція> if <умова>)
```

Вираз-генератор послідовно повертає значення (генерує потік значень), не зберігаючи їх в колекцію. Наприклад, для обчислення суми квадратів парних двоцифрових натуральних чисел замість

```
s = sum([n**2 for n in range(10,100,2)])
```

можна записати

```
s = sum((n**2 for n in range(10,100,2)))
```

або (оскільки круглі дужки, які стосуються виразу-генератора, то можуть бути опущені, якщо він є єдиним аргументом функції) навіть так:

```
s = sum(n**2 for n in range(10,100,2))
```

З генератора завжди можна утворити список, використавши функцію `list()`. Генератор може бути збережений у змінну, але викликатися лише один раз! Це пов'язано з особливостями функціонування ітераторів:[49]

```
s_gen = (n**2 for n in range(10,100,2))
# <generator object <genexpr> at ...>
```

49 У цьому посібнику об'єкти ітераторів детально не розглядаються, для ґрунтовного знайомства з ними звертайтеся до спеціалізованої літератури.

```
s_list_1 = list(s_gen)              # [100, 144, 196, ...]
s_list_2 = list(s_gen)              # []
```

7.1.7. Накладення імен

Змінюваність списків (як і інших змінюваних типів) має побічний ефект, який називають накладенням імен (aliasing). Розглянемо його на прикладі Аліси і Боба, які вирішили вести облік прочитаних книг. Припустимо, що початково вони прочитали ті ж самі книги і вирішили зберегти їх у відповідних списках:

```
aliceBooks = ['Кобзар', 'Енеїда']
bobBooks = aliceBooks
```

Після цього Аліса прочитала ще одну книгу, а Боб вирішив надрукувати свій список прочитаного:

```
aliceBooks.append('Тарас Бульба')
print(bobBooks)       # ['Кобзар', 'Енеїда', 'Тарас Бульба']
```

Прочитана Алісою (але не Бобом) книга потрапила в список прочитаного Боба! Річ у тім, що Боб хотів просто скопіювати список Аліси, який збігався з його списком прочитаного. Але в Python для змінюваних типів операція присвоєння не створює копію об'єкта, а прив'язує нову змінну до того ж об'єкта. Тобто, після виконання коду `bobBooks = aliceBooks` обидві змінні вказуватимуть на той самий об'єкт списку. Щоб уникнути цього, потрібно зробити копію списку, використавши зріз:

```
bobBooks = aliceBooks[:]
```

або функцію `list()`:

```
bobBooks = list(aliceBooks)
```

або метод `copy()`:

```
bobBooks = aliceBooks.copy()
```

або шляхом додавання елементів одного списку у попередньо створений порожній список:

```
bobBooks = []
for book in aliceBooks:
    bobBooks.append(book)
```

Для перевірки, чи змінні вказують на той самий об'єкт, чи на різні об'єкти, призначений оператор `is`. Логічний вираз

```
bobBooks is aliceBooks
```

поверне `True`, якщо обидві змінні вказують на той самий список, і `False`, якщо на різні. Оператор порівняння `==` застосовувати для цього не можна, бо він поверне `True` у випадку двох різних об'єктів списків, які містять однакові елементи.

7.2. Кортежі

Кортеж (тип `tuple`) – це (як і список) впорядкована послідовність (колекція) елементів. Але, на відміну від списків, кортежі незмінювані.

7.2.1. Подання кортежів

Літералом кортежу в Python є послідовність об'єктів, записаних через кому у круглих дужках, наприклад,

```
('зелений', 'жовтий', 'червоний')
# кортеж з рядків - кольорів світлофора
(1, ['Yes', 'No'], 2.1, 'Maybe')
# кортеж з елементів різних типів
()              # порожній кортеж
```

Оскільки круглі дужки в Python використовують також для групування виразів, то, щоб уникнути неоднозначності, кортеж з одного елемента має містити кому після цього елемента:

```
(1,)            # кортеж з одного елемента
```

Кортеж може бути записаний у кілька рядків:

```
rainbow = ('red', 'orange',
```

```
          'yellow', 'green', 'blue',
          'indigo', 'violet')
```

Якщо елементом кортежу є вираз, то інтерпретатор обчислюватиме значення цього виразу:

```
x = 10
print((x-1, x**2-1, x**3-1)) # (9, 99, 999)
```

Кортежі можна об'єднувати, використовуючи оператор "+", але результат об'єднання потрібно перезаписувати:

```
logicalOperators = ('and', 'or')
negate = ('not',)
logicalOperators + negate
print(logicalOperators)             # ('and', 'or')
logicalOperators = logicalOperators + negate
print(logicalOperators)             # ('and', 'or', 'not')
```

Оператор "*" дає змогу заповнювати кортеж однаковими значеннями:

```
print((2.5,)*3)                     # (2.5, 2.5, 2.5)
```

Кортеж також може бути утворений з будь-якої послідовності елементів. Для цього потрібно передати цю послідовність функції `tuple()` як аргумент:

```
l = [1, 2, 3, 4, 5]          # l – список
t = tuple(l)                 # t – кортеж
print(t)                     # (1, 2, 3, 4, 5)
empty = tuple()              # empty – порожній кортеж
```

7.2.2. Індекси та зрізи

Кожен елемент кортежу має індекс – унікальне ціле число, яке однозначно вказує на порядковий номер елемента у кортежі. Індекси можна відраховувати з початку кортежу (тоді нумерація починається з нуля: 0, 1, 2, ...), а можна з кінця (тоді індекси від'ємні: -1, -2, -3 і т. д.):

```
print(rainbow[0], rainbow[-2])              # red indigo
```

Оскільки кортеж – незмінюваний тип даних, то спроба зміни його елемента призведе до помилки[50]:

```
rainbow[2] = 'grey'
# TypeError: 'tuple' object does not support item assignment
```

Проте змінювати елементи змінюваних типів всередині кортежу допускається:

```
t = (1, ['Yes', 'No'], 2.1, 'Maybe')
t[1][1] = 'Yes'
print(t)
# (1, ['Yes', 'Yes'], 2.1, 'Maybe')
```

Якщо елемента з зазначеним індексом не існує, то буде ініційована помилка:

```
print(rainbow[-10])   #IndexError: tuple index out of range
```

Зрізи, як і для списків, дають змогу отримати певну підмножину елементів кортежу. У цьому випадку сам кортеж не “обрізається” (що очевидно через незмінюваність типу), а утворюється новий кортеж. Синтаксис зрізів кортежів, через його повну аналогію з синтаксисом зрізів списків, які розглядали у підрозділі 7.1, тут не наводиться.

7.2.3. Функції для роботи з кортежами та методи кортежів

```
len(rainbow)
# 7 (повертає кількість елементів у кортежі)
rainbow.index('green')
# 3 (повертає індекс елемента)
rainbow.count('green')
# 1 (повертає кількість входжень)
```

50 Саме через незмінюваність кортежів не існує поняття генератора.

7.2.4. Перевірка на входження та поелементний обхід кортежу

Використання оператора перевірки на входження `in` та поелементний обхід кортежів з використанням циклів відбувається аналогічно до відповідних операцій над списками[51].

7.3. Словники

Словник (змінюваний тип `dict`) – це невпорядкована колекція елементів, які організовані асоціативно. Елементи словника називають записами. Запис є парою ключ-значення, тобто доступ до значення конкретного елемента словника відбувається не за його позицією (порядковим номером), як у випадку списків і кортежів, а за ключем.

7.3.1. Подання словників

Літералом словника в Python є послідовність пар ключ-значення, записаних через кому у фігурних дужках, де ключ відділений від значення двокрапкою. Наприклад,

```
person = {'name': 'Роман', 'borned': 1976}
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49,
           8: 64, 9: 81, 10: 100}
emptyDict = {}        # порожній словник
```

Ключами зазвичай є цілі числа або рядки, але теоретично ключем може бути будь-який незмінюваний тип, наприклад, кортеж. Значення можуть бути якого завгодно типу.

Словник також може бути записаний у кілька рядків:

```
pair = {'she': {'name': 'Alice', 'from': 'USA'},
        'he': {'name': 'Bob', 'from': 'Canada'}}
```

51 Див. підрозділ 7.1.

Наведений вище словник `pair` містить два елементи з ключами `'she'` та `'he'`, значення яких є теж словниками.

Якщо ключем або значенням словника є вираз, то інтерпретатор обчислюватиме значення цього виразу:

```
dots1 = {'одна': '.', 'дві': '.'*2, 'три': '.'*3}
print(dots1)  # {'одна': '.', 'дві': '..', 'три': '...'}
```

або

```
dots2 = {'.': 1, '.'*2: 2, '.'*3: 3}
print(dots2)  # {'.': 1, '..': 2, '...': 3}
```

7.3.2. Доступ до елементів словника

Доступ до значень елементів словника відбувається подібно до списків, лише замість індексу у квадратних дужках зазначається ключ:

```
print(squares[6])            # 36
print(pair['he']['from'])    # Canada
```

Оскільки словник – змінюваний тип даних, то його елементи можна змінювати:

```
rectangle = {'length': 2, 'width': 5}
rectangle['width'] = 4
print(rectangle)             # {'length': 2, 'width': 4}
```

Так само можна додати до словника новий елемент (пару ключ-значення):

```
rectangle['square'] = rectangle['length'] * rectangle['width']
print(rectangle)
# {'length': 2, 'width': 4, 'square': 8}
```

Якщо звернутися до елемента словника з неіснуючим ключем, то з'явиться відповідне повідомлення про помилку:

```
print(rectangle['perimeter'])       # KeyError: 'perimeter'
```

Такої помилки можна уникнути, виконавши перевірку за допомогою оператора in:

```
if 'perimeter' in rectangle:
    print(rectangle['perimeter'])
```

Проте ефективнішим вважається використання методу get(), який приймає два аргументи: 1) ключ; 2) значення, яке повертатиметься, якщо у словнику немає елемента з таким ключем:

```
print(rectangle.get('perimeter', 'Даних немає'))
# Даних немає
```

7.3.3. Функції для роботи зі словниками та методи словників

Функція len() повертає кількість елементів словника:

```
points = {'Bob': 91, 'John': 60, 'Alice': 75}
print(len(points))                    # 3
```

Функція sorted() повертає відсортований список ключів словника:

```
print(sorted(points))         # ['Alice', 'Bob', 'John']
```

Крім розглянутого вище методу get(), є ще низка інших (деякі з них часто використовують разом з функцією list)[52]:

```
someDict.pop(key)
# вилучає елемент з вказаним ключем, повертаючи його значення
someDict.clear()              # очищає словник
someDict.copy()               # створює копію словника
someDict.setdefault(key, value)
# якщо ключа key немає у словнику, то додає запис для цього
# ключа із значенням value
list(someDict.keys())         # список ключів
```

52 Отримати повний перелік атрибутів (у тім числі й методів) словників можна, виконавши в інтерактивному режимі команду dir({}).

```
list(someDict.values())      # список значень
list(someDict.items())       # список кортежів (ключ, значення)
```

7.3.4. Поелементний обхід словника

Поелементний обхід словника зазвичай виконують одним з таких двох способів:

```
# Перший спосіб
for key in rectangle:
    print(f'{key} = {rectangle[key]}')

# Другий спосіб
for key,value in rectangle.items():
    print(f'{key} = {value}')
```

В обох випадках буде надруковано:

```
length = 2
width = 4
square = 8
```

7.3.5. Заповнення словників під час виконання програми

Як і списки, на практиці словники часто формуються під час виконання програми. Проте варто пам'ятати, що через неупорядкованість словників, порядок виведених елементів не збігається з порядком долучення елементів до словника. Продемонструємо цей спосіб на прикладі програми, яка обчислює частоту входжень елементів у список:

```
from random import randint
# генеруємо список зі ста 0 і 1:
bits = [randint(0,1) for i in range(100)]
freq = {}                    # створюємо порожній словник
for bit in bits:             # обходимо список поелементно
    if bit not in freq:
        # якщо ключа, який дорівнює елементу, немає у словнику,
```

```
        # то це перше входження (лічильник встановлюємо в 1)
        freq[bit] = 1
    else:                    # інакше
        freq[bit] += 1       # збільшуємо лічильник на 1
print(freq)
# буде виведено на екран словник типу {0: 58, 1: 42}
```

7.3.6. Генератори словників

Словники підтримують генерування шляхом конструкції (інструкція `if` може бути опущена)

```
{ключ: вираз for ключ in колекція if умова}
```

Приклади генерування словника квадратів парних двоцифрових натуральних чисел:

```
d = {n: n**2 for n in range(10,100) if n%2 == 0}
d = {n: n**2 for n in range(10,100,2)}
```

У першому випадку парність числа перевіряється за остачею від ділення на 2, у другому перебір лише парних чисел досягається значенням кроку 2.

7.4. Множини

Множина (змінюваний тип `set`) – це невпорядкований набір (колекція) унікальних (такі, що не повторюються) елементів незмінюваних типів. Крім операцій, характерних для змінюваних структур даних, таких як списки і словники, множини підтримують притаманні їхнім математичним відповідникам операції перетину, об'єднання та різниці.

7.4.1. Подання множин

Літералом множини в Python є послідовність об'єктів, записаних через кому у фігурних дужках:

```
digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
rgbColors = {'red', 'green', 'blue'}# RGB-кольори[53]
assorty = {1, 'one', (1, 1.)} # множина різнотипних елементів
emptySet = set()      # порожня множина[54]
```

Оскільки елементами множини можуть бути тільки незмінювані типи, то спроба створити множину, яка містить список, словник або іншу множину[55], зумовить помилку, наприклад,

```
wrongSet = {1, [2, 3]}   # TypeError: unhashable type: 'list'
```

7.4.2. Операції над множинами

Використовуючи оператори “-”, “|”, “&”, “^”, “>” та ін. Над множинами можна виконувати операції, які не підтримуються іншими колекціями. Розглянемо їх на прикладі двох множин: кольорів моделі RGB і кольорів світлофора:

```
rgbColors = {'red', 'green', 'blue'}
trafficColors = {'red', 'yellow', 'green'}
rgbColors – trafficColors
# {'blue'} (різниця множин)
rgbColors | trafficColors
# {'red', 'green', 'blue', 'yellow'} (об’єднання множин)
rgbColors & trafficColors
# {'red', 'green'} (перетин множин)
rgbColors ^ trafficColors
# {'yellow', 'blue'} (симетрична різниця: елементи, які є
# тільки в одній з множин)
rgbColors > trafficColors
```

53 RGB – модель, яка описує спосіб кодування кольорів.

54 Оскільки {} означає порожній словник, то для порожньої множини введено позначення set().

55 Python дає змогу створити функцією frozenset() спеціальну незмінювану множину, яка може бути елементом множини.

```
# False (перевірка, чи перша множина є надмножиною другої)
rgbColors < trafficColors
# False (перевірка, чи перша множина є підмножиною другої)
'blue' in rgbColors
# True (перевірка на входження оператором in)
```

7.4.3. Функції для роботи з множинами та методи множин

Функція `len()` повертає кількість елементів множини:

```
len(assorty)
# 3 (множина assorty містить ціле число, рядок і кортеж)
```

Функцію `set()`, яка створює множину з послідовності (колекції), часто використовують для видалення дублікатів:

```
from random import randint
bits = [randint(0,1) for i in range(100)]
# буде згенеровано список зі ста нулів і одиниць
print(bits)
# [1, 0, 1, 1, 0, 0, 0, 0, 0, 1, ...
bitSet = set(bits)
# set() утворить зі списка множину, викинувши повторення
print(bitSet)         # {0, 1}
print(list(set('Антарктида')))
# ['р', 'а', 'А', 'н', 'и', 'к', 'т', 'д']
```

Функція `sorted()` повертає відсортований список елементів множини:

```
print(sorted(rqbColors))     # ['blue', 'red', 'yellow']
```

Методи множин:

```
A.add(x)
# додавання елемента x до множини A
A.remove(x)
# вилучення елемента x з множини A (спричиняє помилку у випадку
```

```
# коли немає елемента)
A.discard(x)
# вилучення елемента x з множини A, якщо такий є
A.pop()
# вилучення довільного елемента з множини A
A.clear()
# вилучення всіх елементів (очищення множини A)
A.union(B)
# об'єднання множин A∪B; аналог операції A | B
A.intersection(B)
# перетин множин A∩B; аналог A & B
A.difference(B)
# різниця множин A\B; аналог A – B
A.issubset(B)
# перевірка, чи A є підмножиною B; аналог A < B
```

7.4.4. Генерування множин

Як і розглянуті вище колекції, множини підтримують генерування шляхом конструкції (інструкція `if` може бути опущена)

```
{expression for item in collection if condition}
```

Приклади генерування множини квадратів парних двоцифрових натуральних чисел:

```
s = {n**2 for n in range(10, 100) if n%2 == 0}
s = {n**2 for n in range(10, 100, 2)}
```

На відміну від списків, порядок розташування елементів у множині буде порушений.

Питання та завдання для самоперевірки

1. Що буде виведено на екран в результаті виконання коду?
```
myList = [1, 2, 3, 4, 5, 6]
print(myList[2:-1], myList[0])
```
2. Що буде виведено на екран в результаті виконання коду?
```
l = list('скол')
l.extend([l.pop(), l.pop()])
print(''.join(l))
```
3. Змінити код з підрозділу "Заповнення списків під час виконання програми" так, щоб він формував список унікальних літер, вважаючи велику та малу літеру однією літерою (пробіли і знаки пунктуації у список не включати).
4. Як, не використовуючи цикли і генератори, можна створити нульову матрицю розмірністю 3x3?
5. Чи зміниться значення змінної `x` в результаті виконання коду?
```
x = 'ікс'
y = x
y = 'ігрек'
```
6. Що буде виведено на екран в результаті виконання коду?
```
l1 = l2 = [1, 2]
l2.append(3)
print(sum(l1+l2))
```
7. Чому кортежі не мають притаманних спискам методів `append()` і `pop()`?
8. Що задається виразом `(i+1 for i in range(3))`?
9. Використовуючи кортеж, написати програму, яка виводить назви цифр введеного числа, наприклад, її результатом для числа 100 має бути один нуль нуль.

10. Змінити код з підрозділу "Заповнення словників під час виконання програми" так, щоб умовна інструкція `if/else` стала непотрібною.
11. Який вигляд матиме словник **`myDict`** після виконання коду?
```
myDict = {1: 'один', 'один': 1}
myDict.pop(1)
```
12. Інформація про кількість примірників книг збережена в словнику `library`:
```
library = {'Володимир Малик': {'Князь Кий': 5,
                               'Черлені щити': 1},
           'Макс Кідрук': {'Жорстоке небо': 3,
                           'Зазирни в мої сни': 5,
                           'Навіжені в Перу': 11
                          },
            # і т. д.
          }
```
Написати програму для створення словника, який містить інформацію про загальну кількість примірників книг кожного автора, тобто:
```
{'Володимир Малик': 6,
'Макс Кідрук': 19,
# і т. д.
}
```
13. Що буде виведено на екран у результаті виконання коду? Чому?
```
print(len(set([1, 2, 2, '1', (1, 2)])))
```
14. Скільки елементів міститиме множина s у результаті виконання коду? Чому?

РОЗДІЛ 8
ПРОЦЕДУРНЕ ПРОГРАМУВАННЯ

8.1. Функції та їх використання

До цього часу, якщо не брати до уваги використання вбудованих функцій, програми у попередніх розділах написані в імперативному стилі програмування, тобто складаються з послідовності операцій введення-виведення, інструкцій присвоєння, умовних інструкцій і циклів. Якщо вимагалося виконати ту саму операцію (низку операцій), то її (їхній) код просто повторювався у потрібному місці.

Імперативний стиль зручний лише для невеликих програм. Для більшості прикладних задач він неефективний. Йому на зміну приходить процедурне програмування, в основу якого покладено поняття функції як засобу групування інструкцій для подальшого використання. Функції надають програмістам дві основні переваги:

1) позбавляють необхідності багаторазово повторювати ті самі фрагменти коду, полегшують коригування програм (зміна вноситься один раз у функції, а не кожного разу, де вона використовується);

2) дають змогу розкласти складну задачу на прості частини (декомпозиція).

Якщо у попередніх розділах використовувалися вбудовані або імпортовані функції, то у цьому розглядаються принципи створення власних функцій і різноманітні аспекти їхнього використання.

8.1.1.Створення та виклик функції

Функції прийнято іменувати з малої літери, додаючи між словами-складовими імені символ підкреслення. Для створення функції в Python використовують інструкцію `def`:

```
def <ім’я_функції>(<ім’яАргум1>, <ім’яАргум2>, ...):
    # тіло функції
    <інструкція 1>
```

```
    <інструкція 2>
    ...
```

Тіло функції починає виконуватися в момент виклику функції, синтаксис якого:

```
<ім'я_функції>(<значенняАргум1>, <значенняАргум2>, ...)
```

Викликати функцію можна з будь-якого місця програми (у тім числі й з тіла іншої або навіть тієї ж самої функції[56]), але після інструкції `def`, якою ця функція була створена.

Якщо функція повертає якісь значення, то їх треба зазначати в інструкції `return` у тілі функції. Якщо функція виконує певні операції над об'єктами, але не повертає ніяких значень (в інших мовах програмування такі функції зазвичай називаються процедурами), то інструкція `return` може бути відсутня.

Також прийнято тіло функції починати багаторядковим коментарем (документацією), у якому лаконічно інформується про те, для чого призначена (що робить) ця функція. Для доступу до такої документації функції використовується синтаксис

```
<ім'я_функції>.__doc__
```

Для прикладу розглянемо функцію `average_value`, яка шукає середнє значення елементів списку чисел:

```
def average_value(listOfNumbers):
    """Повертає середнє арифметичне чисел у списку"""
    summa = 0
    for number in listOfNumbers:
        summa += number
    return summa / len(listOfNumbers)
```

Результат виконання цієї функції можна записати у змінну:

```
ages = [25, 30, 35]
```

56 Такі функції називаються рекурсивними і розглядаються у підрозділі 8.2.

```
averageAge = average_value(ages)
print(averageAge)                    # 30.0
```

або вивести безпосередньо:

```
print(average_value([25, 30, 35]))  # 30.0
```

Функція може не мати жодного аргументу, наприклад,

```
def birthday_greeting():
    print('Вітаю з днем народження!')
```

8.1.2. Інструкція return

Під час виклику функції виконання основної програми призупиняється, а керування передається цій функції. Після завершення роботи функція за допомогою інструкції `return` передає результуючі об'єкти назад програмі, яка її викликала.

Якщо інструкції `return` немає, то функція після завершення виконання усіх інструкцій повертає значення `None` (фактично в кінці функції Python неявно додає `return None`), яке зазвичай просто ігнорується основною програмою. Наприклад, розглянута вище функція `birthday_greeting()` просто виводить на екран текст привітання, не передаючи нічого (тобто, передаючи `None`) основній програмі. Перевірити, що функція справді повертає `None`, можна, записавши результат її виконання у змінну – код

```
result = birthday_greeting()
print(result)
```

крім виконаного тіла функції (друку привітання), виведе на екран

```
Вітаю з днем народження!
None
```

Тому результат виконання функцій з відсутньою інструкцією `return` не має змісту записувати у змінну – такі функції зазвичай викликаються безпосередньо. Наприклад,

```
def interchange(l, i, j):
```

```
    """Міняє місцями у списку l елементи з індексами i та j"""
    mem = l[i]              # запам'ятовуємо i-й елемент
    l[i] = l[j]
    # присвоюємо i-му елементу значення j-го елемента
    l[j] = mem
    # присвоюємо j-му елементу запам'ятоване значення
myList = [1, 2, 3, 4, 5]
interchange(myList, 0, 1)
print(myList)               # [2, 1, 3, 4, 5]
```

Інструкція `return` може міститися у будь-якому місці тіла функції. Також у тілі функції можуть міститися кілька інструкцій `return`. Як тільки виконається одна з них, функція припиняє роботу і повертає результат програмі, яка її викликала. Наприклад, у наведеній нижче функції `is_empty_list()`, яка перевіряє, чи список порожній

```
def is_empty_list(someList):
    if someList:            # те саме, що if len(someList) !=0
        return False
    else:
        return True
```

інструкція `else` є зайвою, бо перехід до наступної інструкції `return` відбудеться, лише якщо умова в інструкції `if` не виконається, що відповідає логіці програми. Тобто, умовні інструкції всередині функцій зазвичай записують так:

```
def is_empty_list(someList):
    if someList:
        return False
    return True
```

8.1.3. Повертання більше ніж одного значення

Функція може повертати декілька значень, які подаються через кому після інструкції `return`:

```
def modul(k):
    """Розв'язок рівняння |x| = k"""
    if k < 0:
        return None
    if k == 0:
        return 0
    return -k, k
```

Щоправда, ці значення все одно будуть "запаковані" в один об'єкт – кортеж:

```
solution = modul(5)
print(solution)              # (-5, 5)
```

Для того, щоб отримати конкретні значення, треба звернутися до відповідного елемента кортежу за його індексом:

```
print(solution[0])           # -5
```

8.1.4. Області видимості, локальні та глобальні змінні

Кожного разу, натрапляючи в коді програми на ім'я (змінної, функції тощо), інтерпретатор шукає або створює його в просторі імен. Кожне ім'я має свою область видимості – місце у програмному коді, де цьому імені було присвоєно певне значення.

Будь-які змінні, яким були присвоєні значення всередині функції, враховуючи аргументи функції, з'являються в момент виклику функції і зникають, коли функція завершує роботу і передає керування програмі, яка її викликала. Через цю свою особливість вони називаються локальними змінними. Тобто, кожна функція створює свій локальний простір імен, до якого не можна звернутися ззовні, а кожен виклик функції створює нову локальну область видимості.

Змінні, яким присвоєні значення поза усіма інструкціями `def`, називаються глобальними[57]. Глобальна область видимості охоплює

57 Розрізняють також нелокальні змінні (виникають у випадку вкладених інструкцій def), але у цьому посібнику вони не розглядаються.

єдиний файл (модуль), у якому збережено код. Якщо потрібно використовувати імена з іншого модуля, то його треба явно імпортувати.

Позаяк всередині функції використовується якесь ім'я, то інтерпретатор спочатку шукає його в локальній області видимості інструкції `def`. Не знайшовши цього імені, він виконує пошук в локальних областях видимості охоплюючих інструкцій `def` (якщо такі є), потім у глобальній області видимості (в межах файлу, враховуючи безпосередньо імпортовані модулі), і наостанок у вбудованій області видимості (вбудовані винятки та функції). Якщо в жодній з областей видимості імені не буде знайдено, то з'явиться повідомлення про помилку.

Наприклад, у коді

```
name = 'Роман'
def hi():
    print(f'Привіт, {name}!')
hi()                          # Привіт, Роман!
```

інтерпретатор, не знайшовши значення змінної `name` в локальній області видимості (всередині функції `hi()`), підставляє значення глобальної (визначеної поза інструкцією `def`) змінної з таким самим іменем.

Локальні імена не конфліктують з зовнішніми іменами, навіть якщо вони збігаються:

```
variable = 'global'
def func():
    variable = 'local'
func()
print(variable)               # global
```

У наведеному вище фрагменті коду, не зважаючи на те, що виклик функції `func` змінив значення локальної змінної з іменем `variable`, глобальна змінна з таким самим іменем залишиласть незміненою. Це пояснюється тим, що дві змінні з однаковим іменем `variable` мають різні області видимості.

Якщо функції передається ім'я об'єкта змінюваного типу (список, словник тощо), то функція працює з ним як з глобальним:

```
def add_to_list(item):
    l.append(item)
l = [1, 2, 3, 4, 5]
add_to_list(6)
print(l)              # [1, 2, 3, 4, 5, 6]
```

Щоб дати змогу функції змінювати глобальну змінну незмінюваного типу (число, рядок, кортеж тощо), ім'я цієї змінної оголошується всередині функції інструкцією `global`:

```
name = 'Роман'
def rename(newName):
    global name
    name = newName
rename('Петро')
print(name)           # Петро
```

Без інструкції `global` результат був би таким (значення `'Петро'` присвоюється лише локальній змінній `name` і доступне лише всередині функції, глобальна змінна з таким самим іменем `name` залишається неторканою):

```
name = 'Роман'
def rename(newName):
    name = newName
rename('Петро')
print(name)           # Роман
```

8.1.5. Позиційні та іменовані аргументи. Значення за замовчуванням

За замовчуванням зіставлення аргументів відбувається згідно з їхніми позиціями зліва направо. Під час виклику функції їй треба передати рівно стільки аргументів, скільки імен аргументів зазначено в заголовку

функції, зберігаючи порядок. Якщо функції передати іншу (більшу або меншу) кількість аргументів, то буде згенеровано повідомлення про помилку:

```
from time import sleep
# імпортуємо функцію призупинки виконання програми
def timer(seconds, step):
    """Зворотний відлік часу від seconds секунд з заданим
       кроком step"""
    for second in range(seconds, 0, -step):
        print(second, end = '-')
        sleep(step)
        # призупинка виконання програми на step секунд
    print('stop')
timer(10,1)                  # 10-9-8-7-6-5-4-3-2-1-stop
timer(10)
# TypeError: timer() missing 1 required positional argument:
# 'step'
timer(10,1,2)
# TypeError: timer() takes 2 positional arguments but 3 were
# given
```

Іменовані аргументи дають змогу зіставити аргументи за їхніми іменами. Порядок аргументів під час виклику функції у цьому випадку не має значення:

```
timer(step=2, seconds=10)    # 10-8-6-4-2-stop
```

Можна змішувати позиційні та іменовані аргументи. Тоді спочатку будуть зіставлені зліва направо всі позиційні, а потім іменовані.

Також є можливість задавати значення аргументів за замовчуванням, яке використовуватиметься, якщо під час виклику функції цим аргументам не передаватиметься конкретне значення:

```
def timer(seconds, step = 1):
    """Зворотний відлік часу від seconds секунд з заданим
       кроком step"""
```

```
    for second in range(seconds, 0, -step):
        print(second, end = '-')
        sleep(step)
    print('stop')
timer(10)             # 10-9-8-7-6-5-4-3-2-1-stop
timer(10, 2)          # 10-8-6-4-2-stop
```

У наведеному вище коді аргументу `step` оператором присвоєння встановлене значення за замовчуванням 1. Під час першого виклику функції значення другого аргументу (кроку `step`) не зазначено, тому інтерпретатор призначив йому значення 1. Під час другого виклику аргумент `step` прийняв передане йому значення 2, проігнорувавши значення за замовчуванням.

8.1.6. Передавання довільної кількості аргументів

Функція може приймати довільну кількість аргументів за допомогою параметрів `*args` і `**kwargs`. Перший параметр збирає додаткові позиційні аргументи в кортеж `args`, другий збирає додаткові іменовані аргументи в словник `kwargs`. Насправді імена `args` і `kwargs` просто домовленість, вони можуть бути іншими. Синтаксис визначається лише “зірочками”. Принцип роботи цих параметрів стане зрозумілим з такого коду:

```
def print_friends(person, *friends):
    """Виводить на екран друзів особи person"""
    s = f"{person}'s friends:\n"
    for friend in friends:
        s += friend + ', '
        # додаємо друзів через кому і пробіл
    print(s[:-2] + '.')
    # замінюємо останню кому з пробілом на крапку
def print_info(**kwargs):
    for key in kwargs:
        print(f'{key}: {kwargs[key]}')
```

```
print_friends('Alice', 'Bob', 'Eve', 'John')
# Alice's friends: Bob, Eve, John.
print_info(name = 'Alice', age = 20)
# name: Alice # age: 20
```

Функція `print_friends()` прийняла значення `'Alice'` позиційного аргументу `person`, а решту значень позиційних аргументів зібрала в кортеж `('Bob', 'Eve', 'John')` з іменем `friends`, за елементами якого зробила потім обхід у циклі `for`. Функція `print_info()` приймає довільну кількість іменованих аргументів (у нашому випадку 2), формує з них словник `{'name': 'Alice', 'age': 20}` з іменем `kwargs` і виводить його елементи на екран.

Оскільки параметри `*args` і `**kwargs` працюють з різними типами агрументів (позиційними й іменованими, відповідно), то їх можна використовувати одночасно в межах однієї функції.

8.1.7. Розпаковування аргументів

Якщо `*` (`**`) розмістити перед ітерабельним об'єктом[58] під час виклику функції, то елементи цього об'єкта будуть розпаковані та передані функції як позиційні (іменовані) аргументи, наприклад,

```
def person_info(name, year_of_birth):
    print(f'My name is {name}. I was born in {year_of_birth}.')
my_info_list = ['Roman', 1976]
person_info(*my_info_list)
# My name is Roman. I was born in 1976.
my_info_tuple = ('Roman', 1976)
person_info(*my_info_tuple)
# My name is Roman. I was born in 1976.
my_info_dict = {'year_of_birth': 1976, 'name': 'Roman'}
```

58 Ітерабельний об'єкт – об'єкт, елементи якого можна перебирати по одному (список, словник, кортеж тощо).

```
person_info(**my_info_dict)
# My name is Roman. I was born in 1976.
```

Використання ** у випадку словників забезпечує передавання значень за відповідними ключами. Використання * теж спрацює, але передаватимуться не значення, а ключі, причому в довільному порядку, бо елементи словника, на відміну від елементів списку чи кортежу, не впорядковані:

```
person_info(*my_info_dict)
# My name is year_of_birth. I was born in name.
```

8.1.8. Декомпозиція

Зазвичай функції використовують для декомпозиції задачі. Тобто, задача розбивається на складові частини (підзадачі), для кожної з яких створюється своя функція. Часто функції описуються в окремому файлі (модулі) й імпортуються в основну програму. Це дає змогу багаторазово використовувати функції в різних програмах, імпортуючи їх з потрібного модуля.

Продемонструємо це на прикладі задачі, яка за даними початковими показами виводить покази годинника через заданий проміжок часу. Вважатимемо, що час вводиться і виводиться рядком формату `'години:хвилини.секунди'`. Усі необхідні функції зберігатимемо у модулі (файлі) `funcs.py`.

Розкладемо задачу на складові:

1) введення початкового показу годинника та інтервалу часу (за це відповідатиме функція `input_time()`);

2) перетворення початкового показу годинника в загальну кількість секунд, які пройшли від початку доби (за це відповідатиме функція `time_to_sec()`;

3) перетворення проміжку часу в секунди (за це теж відповідатиме функція `time_to_sec()`;

4) збільшення початкового показу (у секундах) на проміжок часу (теж у секундах); цю звичайну операцію додавання можна теж вкласти (інкапсулювати) у функцію, але не робитимемо цього;

5) перетворення отриманої кількості секунд у формат часу (за це відповідатиме функція `sec_to_time()`);

6) друк отриманого часу.

Файл основної програми[59] міг би мати приблизно такий вигляд:

```
from funcs import *
# імпортуємо усі функції з модуля funcs.py
time = input_time('Введіть початкові покази годинника: ')
interval = input_time('Введіть інтервал часу: ')
initialSeconds = time_to_sec(time)
intervalSeconds = time_to_sec(interval)
totalSeconds = initialSeconds + intervalSeconds
time = sec_to_time(totalSeconds)
print(time)
```

У файлі *funcs.py* мають бути описані всі використані функції:

```
def input_time(text):
    """Запитує рядок 'години:хвилини.секунди' та повертає
       список трьох цілих чисел [години, хвилини, секунди]"""
    time = input(text)
    colonIndex = time.find(':')      # індекс двокрапки
    dotIndex = time.find('.')        # індекс крапки
    return [int(time[:colonIndex]),

int(time[colonIndex+1:dotIndex]),
            int(time[dotIndex+1:])]

def time_to_sec(time):
```

59 Приклад демонстраційний. Звичайно, простіше було би скористатися модулем стандартної бібліотеки datetime.

```
    """На основі списку [години, хвилини, секунди] повертає час
       у секундах"""
    return time[0]*3600 + time[1]*60 + time[2]

def to_str(x):
    """Допоміжна функція, яка перетворює число годин, хвилин
       або секунд у рядок, додаючи символ нуля спереду, якщо
       це число менше 10"""
    if len(str(x)) == 2:
        return str(x)
    return '0' + str(x)

def sec_to_time(secs):
    """Перетворює секунди у рядок 'години:хвилини.секунди'"""
    secs = secs % 86400    # відкидаємо "зайві" секунди, якщо
                           # їхня кількість перевищила добу
    hours = seconds // 3600
    mins = (seconds % 3600) // 60
    secs = (seconds % 3600) % 60
    return f'{to_str(hours)}:{to_str(mins)}.{to_str(secs)}'
```

Частину коду основної програми, яка відповідає за обробку вхідних даних та відображення результату, можна теж вкласти у функцію модуля `funcs.py`, наприклад,

```
def show_new_time(time, interval):
    print(sec_to_time(time_to_sec(time) +
time_to_sec(interval)))
```

Тоді остаточний вигляд програми матиме більш компактний і читабельний вигляд:

```
from funcs import *
time = input_time('Введіть початкові покази годинника: ')
interval = input_time('Введіть інтервал часу: ')
show_new_time(time, interval)
```

8.2. Рекурсивні функції

Деякі задачі шляхом декомпозиції можна звести до підзадач, які розв'язуються за таким самим алгоритмом. Наприклад, щоб обчислити суму елементів списку, треба до першого елемента цього списку додати суму елементів списку, який утворений з попереднього, викинувши перший елемент.

Оскільки алгоритми розв'язання основної задачі та її складових однакові, то для них може бути написана одна єдина функція. Така функція називається рекурсивною.

Особливість рекурсивної функції, порівняно з іншими, в тому, що всередині вона викликає саму себе. Для запобігання нескінченному спрацьовуванню[60] тіло рекурсивної функції має містити принаймні одну умовну інструкцію, в якій перевіряється, чи потрібно продовжувати виконання функції. Наприклад, умовою закінчення підсумовування елементів списку за описаним у першому абзаці цього підрозділу алгоритмом буде досягнення на якомусь етапі порожнього списку (списку з 0 елементів, сума яких, очевидно, 0), а сама функція матиме такий вигляд:

```
def list_sum(L):
    if len(L) == 0:          # можна спростити до if not L:
        return 0
    return L[0] + list_sum(L[1:])
    # до першого елемента додаємо суму решти
```

Використаємо тепер цю функцію для обчислення суми усіх двоцифрових натуральних чисел:

```
myList = [n for n in range(10,100)]
# генеруємо список двоцифрових чисел
mySum = list_sum(myList)
```

60 Насправді жодна рекурсивна функція не виконуватиметься нескінченно. Коли вичерпається ресурс пам'яті, виконання функції перерветься з помилкою RecursionError.

```
# передаємо згенерований список функції
print(mySum)          # виводимо на екран результат: 4905
```

Теоретично будь-який алгоритм, який використовує цикл, може бути перетворений у рекурсивний. Хоча рекурсивний алгоритм виглядає лаконічнішим і природнішим, він зазвичай ресурсозатратніший.

8.3. Анотування функцій

Анотації функцій документують типи аргументів і тип значення, яке повертає функція. Анотування відбувається у заголовку функції. Типи аргументу подають через двокрапку після імені цього аргументу, а тип значення, яке повертається, після символів “->” в кінці заголовка функції, але перед двокрапкою:

```
def count_of_digits(s:str) -> int:
    """Обчислює кількість цифр у рядку"""
    n = 0
    for symbol in s:
        if symbol.isdigit():
            n += 1
    return n

print(count_of_digits('Населення у 2018 р. - 40 млн.'))   # 6
```

Анотація наведеної вище функції інформує про те, що їй передається рядок (тип `str`), а вона повертає ціле число (тип `int`).

Анотування функцій необов’язкове. Анотації описують особливості функції, але не впливають на виконання, позаяк інтерпретатор не перевіряє відповідність типів. Головне призначення анотацій – полегшити користувачам розуміння написаного коду.

8.4. Лямбда-функції

Лямбда-функції – це невеликі функції, які можуть повертати значення лише одного виразу й оголошуються ключовим словом `lambda`. Загальний синтаксис лямбда-функції:

```
lambda аргумент_1, аргумент_2, ..., аргумент_n: вираз
```

Наприклад, лямбда-функцію для обчислення суми, яка буде на рахунку через `years` років у початковому внеску `deposit` і `pc` складних річних відсотках, можна оголосити та викликати так:

```
money = lambda deposit, years, pc: deposit * (1 +
pc/100)**years
my_money = money(1000,2,10)
print(round(my_money,2))            # 1210.0
```

Фактично це скорочена (в один рядок) форма оголошення функції

```
def money(deposit, years, pc):
    return deposit * (1 + pc/100)**years
```

Але є відмінність. Лямбда-функцію не обов'язково називати – вона може бути анонімною (через це лямбда-функції часто називають анонімними функціями) і викликатися у тому ж рядку, що й оголошується:

```
print((lambda deposit, years, pc: deposit * (1 +
pc/100)**years)(1000,2,10))
```

Однак зловживати лямбда-функціями не варто, бо читабельність коду важливіша за його компактність. Зазвичай їх використовують під час відображення функцій на послідовності, фільтруванні та інших специфічних задачах функціонального програмування, які в цьому посібнику не розглядаються[61].

61 Щоб ліпше зрозуміти засоби функціонального програмування, можна опрацювати документацію стосовно функцій map(), filter(), reduce(), zip().

8.5. Оперування функціями як об’єктами

Функції нічим не відрізняються від інших об’єктів Python – вони мають атрибути, їх можна присвоювати змінним, зберігати в структурах даних, передавати аргументами, повертати як результат виконання інших функцій, оголошувати всередині інших функцій тощо. Для демонстрації сказаного оголосимо функцію `new_price`, яка застосовує до старої ціни `old_price` знижку `discount`:

```
def new_price(old_price, discount):
    return old_price * (1 - discount/100)
```

Присвоїмо ім’я функції новій змінній:

```
new_cost = new_price
```

Тепер наступні два рядки коду даватимуть ідентичні результати:

```
print(new_price(50,5))                    # 47.5
print(new_cost(50,5))                     # 47.5
```

У цьому випадку другий виклик працюватиме навіть після вилучення першого імені функції, бо об’єкт функції та її ім’я не те саме:

```
def new_price
print(new_price(50,5))
# NameError: name 'new_price' is not defined
print(new_cost(50,5))                     # 47.5
```

Функція може бути аргументом іншої функції[62], наприклад,

```
def new_price(func, price, percent):
    return func(price, percent)
print(new_price(new_cost,50,5))           # 47.5
```

Розглянемо такий фрагмент коду:

62 На перший погляд це доволі абстрактно. Доцільність передавання функції аргументом іншої функції стане зрозумілою після опрацювання підрозділу “Декоратори”.

```
def apply_discount(percent):
    def discount(price):
        return price * (1 – percent/100)
    return discount
```

По-перше, у ньому всередині функції `apply_discount` оголошено іншу – `discount`. По-друге, результатом виконання функції `apply_discount` є не якесь значення, а функція. Це дає змогу створити функції для конкретної (у відсотках) знижки і надалі передавати їм аргументом початкову ціну:

```
discount_5_percent = apply_discount(5)
discount_10_percent = apply_discount(10)
print(discount_5_percent(50))              # 47.5
print(discount_10_percent(50))             # 45.0
```

Оскільки ім'я вкладеної функції `discount` ніде не фігурує, то функцію `apply_discount` можна переписати простіше з використанням анонімної лямбда-функції:

```
def apply_discount(percent):
    return lambda price: price * (1 – percent/100)
```

Фактично функція `apply_discount` слугує фабрикою для створення інших функцій. Вони можуть бути збережені, наприклад, у списку, і викликатися у циклі:

```
discounts = [discount_5_percent, discount_10_percent]
for func in discounts:
    print(func(50))                        # 47.5   45.0
```

Функції як об'єкти мають атрибути[63]. Крім наперед визначених атрибутів, можна приєднувати і свої. Позаяк атрибути пов'язані з об'єктами, а не з областями видимості, то, хоча вони є локальними стосовно функції, проте зберігають свої значення після виходу з неї.

63 Детальніше атрибути об'єктів розглядатимемо в розділі 10.

8.6. Декоратори

У попередньому підрозділі з'ясували, що функції можуть бути аргументами та результатами виконання інших функцій. Ця можливість покладена в основу функціонування декораторів. Декоратор дає змогу змінювати поведінку функції[64], змінюючи або не змінюючи саму функцію (іншими словами, декоратор підміняє одну функцією іншою). Інтерпретатор Python розпізнає декоратор за символом @. Синтаксис декорування функції виглядає так:

```
@ім'я_декоратора
def ім'я_функції():
    тіло функції
```

Цей запис аналогічний до такого:

```
def ім'я_функції():
    тіло функції
def ім'я_декоратора(функція):
    тіло декоратора
ім'я_функції = ім'я_декоратора(ім'я_функції)
```

Тобто, декоратор – це функція, аргументом якої є інша функція. Результатом виконання декоратора теж має бути функція.

Перший варіант запису, попри свою лаконічність, не завжди ліпший. Оскільки функція декорується безпосередньо під час оголошення, то доступ до оригіналу (недекорованої функції) ускладнюється. Тому інколи доцільніше використовувати другий варіант декорування.

З попередніх фрагментів коду не зрозуміло, що ж міститься всередині функції-декоратора `some_decorator`. Зазвичай там оголошується нова функція-обгортка, не модифікує невідворотно оригінальну функцію, а тільки змінює її поведінку у випадку декорування. Продемонструємо це на конкретному прикладі:

---

64 Декоратор можна застосовувати не тільки до функції, а до будь-якого об'єкта, який можна викликати (методу, класу).

```
# декоратор
def uah(func):
    def wrapper(*args, **kwargs):
        return str(func(*args, **kwargs)) + ' грн.'
    return wrapper
# декорування
@uah
def money(deposit, years, pc):
    return int(deposit * (1 + pc/100)**years)
print(money(1000,2,10))      # 1210 грн.
```

Функцію money вже розглядали раніше. Вона повертає дійсне число, яке дорівнює сумі коштів на рахунку через years років у початковому внеску deposit і pc складних річних відсотках. Функція-декоратор uah змінює її поведінку так, що вона повертатиме цю суму у вигляді рядка, який утворений конкатенацією суми коштів на рахунку та скороченим позначенням української гривні. В обгортці wrapper використано параметри *args і **kwargs, бо декоратору нічого невідомо про аргументи вхідної функції, а він зазвичай має декорувати різні функції з різною кількістю аргументів.

Після декорування функції втрачається прямий доступ до її атрибутів:

```
print(money.__name__)        # wrapper
```

Щоб цього не відбувалося, рекомендується всередині свого декоратора використовувати декоратор functools.wraps стандартної бібліотеки Python, який переносить метадані з недекорованого об'єкта (у нашому випадку – функції) в декорований. Декоратор uah набуде вигляду

```
import functools
def uah(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        return str(func(*args, **kwargs)) + ' грн.'
    return wrapper
```

Тепер після декорування функції доступ до її атрибутів не буде втрачено:

```
print(money.__name__)         # money
```

До функцій можна застосовувати одночасно кілька декораторів, які викликатимуться у зворотному порядку:

```
def decor(func):
    def wrapper(*args, **kwargs):
        return 'Сума коштів: ' +  str(func(*args, **kwargs)) \
               + ' Вітаємо!'
    return wrapper
@decor
@uah
def money(deposit, years, pc):
    return int(deposit * (1 + pc/100)**years)
print(money(1000,2,10))       # Сума коштів: 1210 грн. Вітаємо!
```

Питання та завдання для самоперевірки

1. Чи кожна функція повертає якийсь результат? Чи тільки ті функції, у тілі яких є інструкція `return`?
2. Що відбудеться в результаті виконання коду?
   ```
   print(f(5))
   def f(x):
       return x**x
   ```
3. Скількома порожніми рядками рекомендовано відокремлювати визначення функцій одна від одної згідно з PEP 8?
4. Чому дорівнюватиме значення змінної `l`?
   ```
   l = [2, 1, 3]
   l = l.sort()
   ```

5. Прочитати у підрозділі “Вирази та оператори” розділу 2 про поліморфізм. Подумати, у чому полягає поліморфізм функцій. Навести приклад.
6. Що буде виведено функцією `print()`?
```
def make_5(x):
 if x != 5:
 x = 5
x = 10; make_5(x)
print(x)
```
7. Що буде виведено в результаті виконання коду?
```
def func(person, **info): print(person, info)
func('Alice', age=18, job='student')
```
8. Що буде виведено в результаті виконання коду?
```
def session(algebra, geometry, history=0, english=0):
 print(algebra, geometry, history, english)
session(91, *(66, 82))
```
9. Як, використовуючи розпаковування аргументів, роздрукувати символи рядка **s** (кожен з нового рядка)?
10. Чому функція `is_empty_list()`, описана в підрозділі “Інструкція return”, повертає значення False для списку `emptyList = [[]]`?

РОЗДІЛ 9
ЗАСОБИ ОПРАЦЮВАННЯ ВИНЯТКІВ

Якісна програма має передбачати можливість виникнення помилок виконання і правильно реагувати на них, а не завершуватись аварійно зі стандартними повідомленнями про помилку. Для реалізації цього в Python вбудований механізм опрацювання винятків (`exception handling`), який ґрунтується на тому, що коли під час виконання синтаксично коректної програми трапляється помилка, інтерпретатор генерує виняток. Ці винятки, які викликають збій програми, можна контролювати. Крім контролю за помилками, опрацювання винятків використовується для гарантії виконання певних заключних операцій і для повідомлення про настання певних подій.

9.1. Основні типи винятків

Інтерпретатор Python підтримує велику кількість вбудованих типів винятків[65], серед яких:

`FileNotFoundError` (виникає у разі спроби відкрити неіснуючий файл);

`ImportError` (у разі спроби імпорту неіснуючого модуля);

`IndexError` (у разі спроби доступу до елемента з неіснуючим індексом);

`KeyError` (у разі спроби доступу до елемента словника з неіснуючим ключем);

`NameError` (у разі спроби доступу до неіснуючого об'єкта);

`TypeError` (у разі спроби виконати недоступну операцію)

`ZeroDivisionError` (у разі спроби виконання ділення на 0).

65 Крім вбудованих типів винятків, існує також можливість визначати власні, користувацькі типи.

9.1.1. Інструкція try

Для опрацювання винятків використовують інструкцію `try`[66], найпростіший синтаксис якої складається з двох блоків: власне `try` і блоку `except`:

```
try:
    # цей блок виконується завжди
    ...
except:
    # цей блок виконується, якщо у блоці try трапиться виняток
    ...
```

Тобто, код, під час виконання якого можуть трапитися помилки виконання (винятки), вкладається в блок `try`, а блок `except` містить код, який виконуватиметься тільки коли в блоці `try` трапиться виняток (неважливо якого типу). Блок `except` запобігає аварійному завершенню програми, а після його виконання (або невиконання в разі успішного виконання блоку `try`) керування завжди передається наступній (після конструкції `try/except`) інструкції.

У блоці `except` можна зазначати конкретний тип винятку, крім того, таких блоків може бути кілька:

```
try:
    # цей блок виконується завжди
    ...
except <ТипВинятку1>:
    # цей блок виконується, якщо у блоці try трапиться
    # виняток першого типу
    ...
except <ТипВинятку2>:
    # цей блок виконується, якщо у блоці try трапиться
    # виняток другого типу
```

66 Опрацюванню винятків стосуються також інструкції raise і assert (тут не розглядаються).

```
    ...
...
except:
    # цей необов'язковий блок виконується, якщо у блоці try
    # трапиться виняток будь-якого типу, іншого від
    # перерахованих у попередніх блоках except
    ...
```

Розглянемо процес використання механізму опрацювання винятків на прикладі програми, яка обчислює суму чисел, записаних у деякому текстовому файлі по одному числу в рядку:

```
fileName = input("Введіть ім'я файлу: ")
with open(fileName) as f:    # відкриття файлу в режимі читання
        lines = f.readlines() # формування списку рядків файлу
numbers = list(map(float, lines))
# перетворення списку рядків у список чисел
print(sum(numbers))          # виведення суми елементів списку
```

Ця програма передбачає принаймні таке:

1) файл з введеним іменем існує;

2) у кожному рядку файлу записане число.

Якщо хоча б один пункт не виконується, то виконання програми призведе до аварійного завершення з повідомленням про відповідну помилку[67]:

```
FileNotFoundError: [Errno 2] No such file or directory або
ValueError: could not convert string to float.
```

Тому для коректного виконання програми команди, для яких існує ризик помилки, треба вкласти в інструкцію `try`:

```
fileName = input("Введіть ім'я файлу")
try:
    with open(fileName) as f:
```

67 Насправді можуть трапитися й інші винятки, наприклад, PermissionError у випадку, коли файл захищений від читання.

```
        lines = f.readlines()
        numbers = list(map(float, lines))
except FileNotFoundError:
    print('Файлу з заданим іменем не знайдено!')
except ValueError:
    print('Файл містить нечислові дані!')
else:
    print(sum(numbers))
```

Крім блоків `except`, інструкція `try` може мати ще два необов'язкові блоки: `else` і `finally`. Перший виконується, тільки якщо блок `try` виконався без винятків, а другий завжди, чим гарантує виконання принципово важливих заключних операцій незалежно від того, трапилися винятки чи ні.

Питання та завдання для самоперевірки

1. Що буде з програмою у випадку настання винятку, якщо не передбачити його опрацювання?
2. Подумати, яка функція менеджера контексту `with` у наступному коді (вважаючи, що файл *filename.txt* існує, а винятки можуть трапитись тільки в блоці інструкцій):
   ```
   with open('filename.txt') as file:
       # блок інструкцій
   ```
   Написати аналог цього коду, використавши відповідний варіант інструкції **`try`**.

РОЗДІЛ 10
ВСТУП В ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ НА PYTHON. ОСНОВНІ ЗАСАДИ ТА ПРИНЦИПИ

Об'єктно-орієнтоване програмування (ООП) – одна з парадигм програмування, яка розглядає програму як множину "об'єктів", які взаємодіють між собою. Головно ООП використовують, коли програмні продукти передбачають довгострокову розробку і супровід.

Як і у процедурному програмуванні, в ООП програма розкладається на складові. Якщо ці складові використовуються у нових програмах, то вони зазвичай не видозмінюються, а адаптовуються.

Попри те, що раніше в цьому посібнику ми вживали термін "об'єкт", а програми використовували об'єкти у виразах, передавали їх функціям, викликали їхні методи тощо, стиль програмування не був об'єктно-орієнтованим. Для того, щоб він став таким, об'єкти мають бути учасниками ієрархічного успадкування. Крім успадкування, для ООП характерні поліморфізм, композиція та інкапсуляція.

10.1. Поняття клас та екземпляр класу

В ООП розрізняють два різновиди об'єктів – класи й екземпляри. Класи реалізують поведінку і можливість створення екземплярів, а екземпляри – власне об'єкти, які опрацьовує програма.

Класи – це програмні компоненти для реалізації нових типів об'єктів, які характеризуються станом і поведінкою та підтримують успадкування. Вони є відображенням об'єктів реального світу в програмному коді та слугують своєрідними фабриками об'єктів. Кожного разу під час виклику класу створюється новий об'єкт (екземпляр цього класу), який, крім доступу до всіх атрибутів класу, має свій власний простір імен для своїх даних, відмінних від даних інших екземплярів цього самого класу. Цим класи чимось нагадують модулі з функціями, але, на відміну від них, мають свої особливості, основними з яких є підтримка

створення багатьох екземплярів, успадкування та перевантаження операторів.

Класи прийнято іменувати, починаючи з великої літери, у цьому випадку слова-складові імені теж починаються з великої літери, а символи підкреслення не використовуються. Для створення класу використовують складену інструкцію `class`:

```
class <Ім'яКласу>:
    тіло класу
```

Наприклад, створимо клас `Writing`, який у майбутньому репрезентуватиме літературний твір:

```
class Writing():
    pass     # порожня інструкція, яка резервує місце
             # під тіло класу
```

На відміну від класів, які створюються інструкціями, екземпляри створюються викликами:

```
<ім'яЕкземпляру> = <Ім'яКласу>()
```

Незважаючи на те, що клас `Writing` на цьому етапі порожній, є можливість створювати екземпляри цього класу:

```
novel = Writing()
story = Writing()
```

У реальних програмних продуктах класи часто розміщують в окремих файлах (модулях), які потім імпортуються до файлу основної програми. Якщо опис класу `Writing` міститься у файлі *myclasses.py*, то інстанціація екземпляра в файлі основної програми набуде вигляду:

```
import myclasses
novel = myclasses.Writing()
```

або

```
from myclasses import Writing
novel = Writing()
```

10.2. Атрибути та методи класу

Атрибути класу описують стан екземплярів, а методи – притаманну їм поведінку. Атрибути (змінні класу) створюються інструкціями присвоєння, а методи (функції, визначені всередині класу) – вкладеними в інструкцію `class` інструкціями `def`. Після виконання інструкції `class` усі атрибути та методи стають доступними через точкову нотацію, у цьому випадку кожен екземпляр успадковує всі атрибути та методи класу:

```
об’єкт.атрибут              # доступ до атрибута
об’єкт.атрибут()            # доступ до методу
```

Операція присвоєння створює атрибут класу, спільний для всіх майбутніх екземплярів цього класу. Щоб зазначити, що має створитися атрибут, значення якого різні для різних екземплярів, як ім’я об’єкта використовується `self`[68]. Крім того, `self` також завжди передається першим аргументом у заголовках методів класу.

Розширимо наведений вище клас `Writing`, додавши до нього два методи `set_data()` і `print_data()` для задання та виведення на екран атрибутів (автора і назви твору), відповідно:

```
class Writing():                    # власне створення класу

    info = 'Літературний твір'      # створення атрибута класу

    def set_data(self, author, title):
    # створення методу класу
        self.author = author        # створення атрибута
        self.title = title          # створення атрибута
    def print_data(self):           # створення методу класу
        print(f'{self.author}. "{self.title}"')# друк атрибутів
```

68 Іноді self називають псевдонімом об’єкта.

Щоб метод `set_data()` зміг присвоїти певні значення атрибутам `author` (автор) і `title` (назва твору), ці значення мають бути передані йому через аргументи (як зазначалося раніше, першим аргументом завжди є `self`). Наприклад, в інструкції `self.author = author` точкова нотація `self.author` позначає атрибут `author` екземпляра класу, а `author` – значення, яке передається аргументом методу `set_data()` і буде присвоєне цьому атрибуту.

Тобто, змінна `author` як локальна змінна доступна лише всередині методу `set_data()`, а атрибут `self.author` доступний з будь-якого місця всередині класу. Це пояснює непотрібність передавання аргументів методу `print_data()` для їх виводу на екран, бо після виконання інструкцій присвоєння у `set_data()` їхні значення стануть атрибутами класу, і, відповідно, усіх екземплярів цього класу.

Для демонстрації функціональності класу `Writing` створимо екземпляр, який репрезентуватиме конкретний літературний твір, наприклад, “Захар Беркут” Івана Франка:

```
zb = Writing()
zb.set_data('Іван Франко', 'Захар Беркут')
zb.print_data()              # Іван Франко. "Захар Беркут"
```

Тепер, використовуючи точкову нотацію, можна отримати безпосередній доступ до атрибутів екземпляра:

```
# отримання значення атрибута екземпляра
print(zb.title)              # Захар Беркут


# зміна атрибута
zb.author = 'І. Франко'
zb.print_data()              # І. Франко. "Захар Беркут"
```

У цьому випадку атрибут класу info буде спільний для всіх екземплярів:

```
tm = Writing()
print(zb.info)               # Літературний твір
print(tm.info)               # Літературний твір
```

Змінювати такі атрибути потрібно через об'єкт класу:

```
Writing.info = '--Літературний твір--'
print(zb.info)        # --Літературний твір--
print(tm.info)        # --Літературний твір--
```

Якщо змінити атрибут класу через об'єкт екземпляра, то зміниться значення атрибута тільки для цього екземпляра, а значення атрибута класу (і інших екземплярів цього класу) залишиться незмінним:

```
zb.info = 'Літературний твір'
print(zb.info)        # Літературний твір
print(tm.info)        # --Літературний твір--
```

Кожен клас має набір спеціальних атрибутів, імена яких починаються і закінчуються двома символами підкреслення, наприклад,

```
__dict__()            # словник простору імен класу
__class__()           # посилання на суперклас[69] класу
__bases__()           # повертає кортеж суперкласів класу
```

Існує також можливість створення нових атрибутів екземпляру:

```
zb.year = 1882 # рік написання,
```

але у такій формі він зазвичай не використовується, бо новостворені таким способом атрибути не доступні методам класу без ручного оновлення. Щоб такий атрибут став атрибутом усіх наступних створених екземплярів класу `Writing`, у тіло методу `set_data()` потрібно додати рядок

```
self.year = year
```

10.2.1. Конструктор класу

В ООП імена методів Python-класів, які починаються і закінчуються двома символами підкреслення, мають спеціальне призначення. Одним з таких методів є метод `__init__()`, який називається конструктором

69 Для розуміння поняття "суперклас" потрібно опрацювати підрозділ 10.4.

класу і викликається автоматично в момент створення екземпляра. Мета конструктора – ініціалізація початкових значень атрибутів.

У попередньому прикладі новостворений екземпляр `zb` класу `Writing` був "порожнім", він набував змісту тільки після задання конкретних значень його атрибутам викликом методу `set_data()`. Зазвичай перші значення присвоюються атрибутам у конструкторі класу. Оскільки конструктор викликається автоматично, то, змінивши ім'я `set_data` на `__init__`, ми одним рядком зможемо створити "повноцінний" (з початковими значеннями атрибутів) екземпляр:

```
class Writing():

    def __init__(self, author, title):      # конструктор
        self.author = author
        self.title = title

    def print_data(self):
        print(f'{self.author}. "{self.title}"')

zb = Writing('Іван Франко', 'Захар Беркут')
# інстанціація екземпляра
zb.print_data()              # Іван Франко. "Захар Беркут"
```

Початкові значення значень атрибутів екземпляра передаються у круглих дужках після імені класу під час інстанціації екземпляра у тому ж порядку, в якому наведений перелік цих атрибутів в заголовку конструктора після першого обов'язкового аргументу `self`[70].

70 У методах класу можна використовувати й інші способи передавання аргументів.

10.2.2. Метод __str__()

У разі виведення на екран значення екземляра функцією `print()` відобразиться інформація про відповідний об'єкт та його розташування в пам'яті комп'ютера, наприклад,

```
print(zb)      # <__main__.Writing object at 0x7f1ffee53198>
```

Для виведення змістовної інформації про екземпляр у класі `Writing` був описаний метод `print_data()`. Щоб забезпечити таку саму функціональність для вбудованої функції `print()`, використовують спеціальний метод `__str__()`, який інструкцією `return` повертає рядок у потрібному для відображення форматі[71]. Замінимо у класі `Writing` метод `print_data()` на `__str__()`[72]

```
class Writing():

    def __init__(self, author, title):
        self.author = author
        self.title = title

    def __str__(self):
        return f'{self.author}. "{self.title}"'
```

і продемонструємо його функціонал на прикладі екземпляра `zb` з попереднього підрозділу:

```
print(zb)             # Іван Франко. "Захар Беркут"
```

Тобто, метод `__str()__` призначений для рядкового подання інформації про стан об'єкта (формує рядок зі значеннями атрибутів об'єкта у зручній формі), яке за замовчуванням передається аргументом функції `print()` у разі спроби надрукувати цей об'єкт.

---

71 Для задання формату відображення у вікні інтерактивної оболонки замість методу __str__() використовують метод __repr__().

72 Метод print_data() можна було залишити, це не принципово.

10.3. Інкапсуляція

Над екземплярами (атрибутами екземплярів) можна виконувати різні операції. Наприклад, можна замінити ім'я автора на ініціал:

```
nameList = zb.author.split()
# утворюємо список з імені та прізвища
nameList[0] = nameList[0][0] + '.'
# замінюємо ім'я на ініціал (з крапкою)
zb.author = ' '.join(nameList)
# переприсвоюємо значення атрибута
print(zb)
# I. Франко. "Захар Беркут"
```

На практиці атрибути екземплярів зазвичай не змінюють безпосередньо, а використовують інкапсуляцію – концепцію проектування, основна ідея якої полягає в реалізації операцій над об'єктами у вигляді методів класу. У нашому випадку доцільно додати до класу `Writing` метод `transform()`, який перетворює ім'я автора в ініціал (у цьому методі відбувається перевірка, чи не є ім'я автора вже ініціалом):

```
class Writing():
    def __init__(self, author, title):
        self.author = author
        self.title = title
    def __str__(self):
        return f'{self.author}. "{self.title}"'
    def transform(self):
        words = self.author.split()
        if words[0][-1] != '.':
            words[0] = words[0][0] + '.'
            self.author = ' '.join(words)
```

Тоді ця операція виконуватиметься не безпосередніми перетвореннями, а викликом методу `transform()`:

```
tm = Writing('Олександр Дюма', 'Три мушкетери')
```

```
print(tm)              # Олександр Дюма. "Три мушкетери"
tm.transform()
print(tm)              # О. Дюма. "Три мушкетери"
```

Отже, інкапсуляція зосереджує код в одному місці, не розпорошуючи його по всій програмі, що дає змогу застосовувати операції до усіх екземплярів і полегшує супровід програмного забезпечення.

10.4. Успадкування

Механізм успадкування забезпечує можливість зміни функціональності програм не шляхом зміни існуючого програмного коду, а створюючи нові програмні компоненти підкласів, які успадковують атрибути і методи батьківських класів, можуть модифікуватися та розширюватися. У цьому випадку зміни у підкласах не зачіпають суперкласи.

Для успадковування атрибутів іншого класу достатньо зазначити ім'я цього класу в круглих дужках у заголовку інструкції `class`. Клас, який успадковує, називається підкласом (або дочірнім класом), а клас, від якого успадковуються атрибути та методи – суперкласом (або батьківським класом). Екземплярам підкласу стають доступні атрибути і методи суперкласу та підкласу. Якщо ці атрибути чи методи однойменні, то перевага надається тим, які належать підкласу. Це чимось схоже на області видимості функцій – якщо функція не знаходить імені всередині себе, то вона починає шукати в охоплюючій її функції чи програмі. У випадку ООП пошук імен відбувається по ієрархічному дереву класів аж до його верхівки[73].

Для демонстрації механізму успадкування створимо на базі класу `Writing` клас `Publication`, який репрезентує друковане видання

73 У випадку успадкування від кількох суперкласів порядок пошуку визначається порядком, у якому зазначені їхні імена (через кому у круглих дужках в заголовку інструкції class).

літературного твору. Крім атрибутів і методів, успадкованих від свого суперкласу, він матиме ще низку характерних для видання атрибутів: видавництво та рік видання.

```
class Publication(Writing):
    counter = 0       # лічильник екземплярів (атрибут класу)
    def __init__(self, author, title, edition, year):
        # конструктор
        Writing.__init__(self, author, title)
        # виклик конструктора суперкласу
        self.edition = edition
        # видання (атрибут екземпляра)
        self.year = year     # рік видання (атрибут екземпляра)
        Publication.counter += 1
        # збільшуємо лічильник екземплярів на 1
    def __str__(self):
        return Writing.__str__(self) + \
               f'("{self.edition}", {self.year})'
```

Щоб показати принцип застосування атрибутів класу, спільних для усіх екземплярів, у класі `Publication` передбачено лічильник екземплярів `counter`. У конструкторі підкласу `Publication` викликано конструктор його суперкласу `Writing`, щоб створити атрибути `author` і `title`, а потім створено додаткові атрибути `edition` (видавництво) і `year` (рік видання) та виконано збільшення лічильника екземплярів на одиницю (позаяк конструктор автоматично запускається під час інстанціації екземпляра). Аналогічно у методі `__str__()` результат однойменного методу суперкласу конкатенується з додатковою інформацією, характерною підкласу, хоча можна було би сформувати рядок для відображення заново, наприклад,

```
    def __str__(self):
        return f'{self.author}. "{self.title}" \
                ("{self.edition}", {self.year})'
```

Перевіримо функціональність підкласу Publication:

```
pub_1 = Publication('О. Дюма', 'Три мушкетери', 'Фоліо', 2005)
pub_2 = Publication('Іван Франко', 'Захар Беркут',
                    'Каменяр', 2002)
print(pub_1)  # О. Дюма. "Три мушкетери"("Фоліо", 2005)
print(pub_2)  # Іван Франко. "Захар Беркут"("Каменяр", 2002)
```

Переконаємось, що успадкувався описаний в класі `Writing` метод `transform()`, тобто, що він доступний з екземплярів класу `Publication`:

```
pub_2.transform()
print(pub_2)  # І. Франко. "Захар Беркут"("Каменяр", 2002)
```

Лічильник екземплярів також функціонує:

```
print(Publication.counter)   # 2
```

Якщо функціональність деякого методу суперкласу не задовольняє потреб підкласу, то можна виконати заміщення методів, тобто написати у підкласі новий метод з таким самим іменем. У цьому випадку доступ до однойменного методу суперкласу не втратиться, його можна буде викликати так само, як у конструкторі класу `Publication` викликався конструктор його суперкласу `Writing`.

Python також підтримує множинне успадкування, тобто успадкування від кількох класів, загальний синтаксис якого

```
class І'мя_класу(Суперклас_1, Суперклас_2, ...):
    # тіло класу
```

10.5. Композиція

Композиція в ООП – спосіб відображення взаємовідношень між об'єктами, який полягає у вбудовуванні в клас-контейнер об'єктів інших класів.

Клас `Publication` має суттєве обмеження. Він репрезентує видання, яке містить тільки один літературний твір. Щоб мати змогу репрезентувати збірку творів, створимо клас `Book`, який

використовуватиме композицію, тобто міститиме екземпляри творів-складових, які ми подаватимемо новим, успадкованим від `Writing`, класом `Item` з додатковим атрибутом `size` (обсяг у сторінках):

```
class Item(Writing):
    def __init__(self, author, title, size):
        Writing.__init__(self, author, title)
        self.size = size

    def __str__(self):
        return Writing.__str__(self) + f' ({self.size} стор.)'
class Book():
    def __init__(self, edition, year, items = []):
        self.items = items
        self.edition = edition
        self.year = year

    def add_item(self, item):
        self.items.append(item)

    def total_size(self):
        totalSize = 0
        for item in self.items:
            totalSize += item.size
        return totalSize

    def __str__(self):
        s = f'Видавництво "{self.edition}", {self.year} рік\n'
        s += f'Загальний обсяг: {self.total_size()} стор.\n'
        s += '-'*50
        for item in self.items:
            s += f'\n{item.__str__()}'
        return s
```

У конструкторі класу Book зазначено, що інстанціація екземпляра відбувається на основі списку літературних творів items. Значення за замовчуванням [] забезпечує створення порожньої збірки творів, якщо списку не буде передано. Для додавання твору у збірку призначений метод add_item(). Також описано метод total_size(), який повертає загальний обсяг збірки. Метод __str__() формує рядок, який відображає інформацію про збірку у зручному для сприйняття вигляді.

У результаті виконання коду, який створює два літературні твори і додає їх до збірки:

```
item_1 = Item('О. Дюма', 'Три мушкетери', 654)
item_2 = Item('І. Франко', 'Захар Беркут', 120)
book = Book('Самвидав', 2018, [item_1, item_2])
print(book)
```

буде виведено на екран

```
Видавництво "Самвидав", 2018 рік
Загальний обсяг: 774 стор.
--------------------------------------------------
О. Дюма. "Три мушкетери" (654 стор.)
І. Франко. "Захар Беркут" (120 стор.)
```

Після додавання третього твору до збірки

```
item_3 = Item('Л. Українка', 'Лісова пісня', 76)
book.add_item(item_3)
print(book)
```

буде виведена інформація про оновлений екземпляр:

```
Видавництво "Самвидав", 2018 рік
Загальний обсяг: 850 стор.
--------------------------------------------------
О. Дюма. "Три мушкетери" (654 стор.)
І. Франко. "Захар Беркут" (120 стор.)
Л. Українка. "Лісова пісня" (76 стор.)
```

10.6. Перевантаження операторів

Перевантаження операторів дає змогу екземплярам класів бути складовими виразів, тобто брати участь у різних операціях (арифметичних, порівняння, доступ до елемента за індексом, зрізи тощо). Для кожного конкретного оператора зарезервоване спеціальне ім'я методу, наприклад,

```
__add__()             # оператор "+"
__sub__()             # оператор "-"
__mul__()             # оператор "*"
__div__()             # оператор "/"
__mod__()             # оператор "%"
__eq__()              # оператор "=="
__ne__()              # оператор "!="
__lt__()              # оператор "<"
__le__()              # оператор "<="
__gt__()              # оператор ">"
__ge__()              # оператор ">="
__getitem__()         # оператор "[]"
```

Операції, які виконуються з використанням таких операторів, еквівалентні викликам відповідних методів, наприклад, `x + y` те саме, що `x.__add__(y)`.

Якщо клас не перевантажує якогось оператора (чи не успадковує методу перевантаження цього оператора), то це означає, що екземпляри класу не підтримують відповідну операцію, а спроба її виконати викличе помилку.

Для прикладу перезавантажимо оператор "+" у класі `Book` так, щоб він додавав до збірки літературний твір, тобто діяв аналогічно до методу `add_item()` і оператор порівняння **"=="** так, щоб він порівнював дві різні збірки, повертаючи `True`, якщо вони однакові за обсягом і `False`, якщо ні (у коді класу наведено тільки методи перевантаження операторів, решта методів – `__init__()`, `add_item()`, `total_size()` і `__str__()` – залишаються такими самими і тому опущені):

```
class Book():

    def __add__(self, item):
        self.items.append(item)

    def __eq__(self, book):
        return self.total_size() == book.total_size()
```

Нижче наведено результати виконання операцій з екземплярами класів, які використовують перезавантажені оператори “+” і “==” (екземпляр book взято з попереднього підрозділу):

```
another_book = Book('Книголюб', 2019)
# створюємо порожню збірку
another_book + item_1 # додаємо твір до збірки
print(another_book)
# Видавництво "Книголюб", 2019 рік
# Загальний обсяг: 654 стор.
# --------------------------------------------------
# О. Дюма. "Три мушкетери" (654 стор.)
print(book == another_book)
# False (збірки не однакові за обсягом)
another_book + Item('М. Гоголь', 'Вій', 196)
# додаємо ще один твір
print(another_book)
# Видавництво "Книголюб", 2019 рік
# Загальний обсяг: 850 стор.
# --------------------------------------------------
# О. Дюма. "Три мушкетери" (654 стор.)
# М. Гоголь. "Вій" (196 стор.)
print(book == another_book)
# True (тепер збірки однакові за обсягом)
```

Якщо спробувати виконати над класами операцію, для якої не описано метод перевантаження відповідного їй оператора, наприклад,

```
another_book - item_1
```

то з'явиться повідомлення про помилку:

```
TypeError: unsupported operand type(s) for -: 'Book' and 'Item'
```

Питання та завдання для самоперевірки

1. До якого типу мов програмування, процедурних чи об'єктно-орієнтованих, належить Python?
2. Що з наведеного найбільше підходить для імені класу?
   `my_class` `myClass` `my_Class` `MyClass` `My_Class` `My_class`
3. Який метод класу автоматично викликається в момент створення екземпляра цього класу?
4. На яку операція з процедурного програмування синтаксично схожа інструкція створення екземпляра класу?
5. Як можна розширити функціональність успадкованого від суперкласу методу, не переписуючи його повністю заново?
6. В окремому модулі створити клас, який репрезентує вектор за двома координатами. Перевантажити оператор "*" так, щоб він повертав скалярний добуток векторів. Написати програму, яка обчислює скалярний добуток двох векторів.
7. Вважаючи створений у попередньому завданні клас суперкласом, створити дочірній клас такої самої функціональності, але для тривимірних векторів.

РОЗДІЛ 11
ГРАФІЧНІ ІНСТРУМЕНТИ ТА ЗАСОБИ PYTHON

11.1. Графічний інструментарій

Графічний інструментарій Python дає змогу відображати геометричні форми у дво- та тривимірному просторі. Для створення простої графіки використовують модуль `turtle` стандартної бібліотеки, а для більш спеціалізованої – сторонні пакети[74] (cairo, matplotlib, plotly та ін), які потребують додаткової інсталяції.

11.1.1. Модуль turtle

Модуль `turtle` забезпечує простий інтуїтивно зрозумілий підхід до рисування у вікні, який зводиться до керуванням об'єктом-черепахою (від англ. `turtle` – черепаха), до хвоста якої причеплений пензлик. Керування в найпростішому випадку полягає в піднятті чи опусканні хвоста, руху вперед і повороту. Якщо чепераха рухається у вікні з опущеним хвостом, то вона залишає за собою слід.

У будь-який момент стан черепахи описується набором атрибутів, серед яких:

1) `shape` – геометрична форма, якою відображається черепаха у вікні (за замовчуванням вістря стрілки);

2) `position` – положення (x, y) в декартовій системі координат, початок якої у центрі вікна; початкове положення: (0; 0);

3) `heading` – кут у градусах між віссю абсцис і напрямом черепахи, який відраховується проти годинникової стрілки (початкове значення 0 градусів, тобто черепаха скерована вправо);

4) `color` – колір пензлика (за замовчуванням чорний);

5) `width` – ширина пензлика (за замовчуванням 1 піксель);

74 Пакет – ієрархічний набір модулів.

6) down – атрибут логічного типу, який зазначає, чи опущений хвіст черепахи, тобто, чи залишатиме вона за собою слід під час руху (за замовчуванням True).

Інтерфейс, через який програміст керує черепахою (екземпляром t класу Turtle), описується набором методів[75]:

```
# методи зміни атрибутів черепахи
t.up()
# піднімає хвіст (пензлик)
t.down()
# опускає хвіст (пензлик)
t.setheading(z)
# надає атрибуту heading значення z градусів
t.left(z)
# повертає черепаху на z градусів проти годинникової стрілки
t.right(z)
# повертає черепаху на z градусів за годинниковою стрілкою
t.pencolor(s)
# встановлює колір пензлика (значення аргументу s – це рядки
# типу 'red', 'green', 'blue' та ін.)[76]
t.fillcolor(s)
# встановлює колір зафарбовування (якщо значення кольору не
# передати явно, зафарбовуватиметься кольором пензлика)
t.begin_fill()
# зазначає початок рисування фігури, яка зафарбовуватиметься
t.end_fill()
# зафарбовує фігуру, яка нарисована командами після
# t.begin_fill()
t.screen.bgcolor(s)
# встановлює колір фону вікна
```

75 Наведений перелік методів не детальний.

76 Функції можна передавати три атрибути, які відповідають RGB-значенню кольору – pencolor(r,g,b).

```
t.width(p)
# встановлює ширину пензля рівною p пікселів
t.hideturtle()
# приховує зображення черепахи (вістря стрілки) у вікні

# методи перевірки стану
t.position()
# повертає координати черепахи у вигляді кортежу (x,y)
t.heading()
# повертає значення напряму (атрибут heading) у градусах
t.isdown()
# повертає True, якщо хвіст опущений і False, якщо піднятий
# методи переміщення черепахи
t.home()
# переміщує черепаху в початкове положення
t.goto(x,y)
# переміщує черепаху в положення з координатами (x,y)
t.forward(d)
# переміщує черепаху на d пікселів у напрямі heading
```

Розглянемо функціональність модуля `turtle` на прикладі коду, який рисує фігуру, зображену на рис. 11.1.

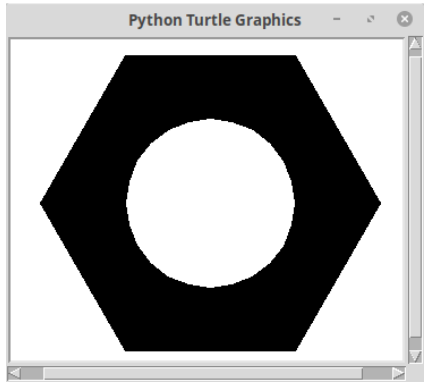


Рис. 11.1. Рисунок Python Turtle Graphics

```
import turtle
t = turtle.Turtle()
# створення екземпляра черепахи і вікна для рисування
# Переміщення черепахи у крайню праву вершину шестикутника
t.up()
# оскільки шестикутник замальовуватиметься, то піднімати
# пензлик не обов'язково
t.forward(150)              # пересуваємо черепаху
t.left(120)                 # і скеровуємо її вздовж сторони
t.down()                    # опускаємо пензлик
# Рисуємо шестикутник
t.begin_fill()
# зазначаємо, що шестикутник зафарбовуватиметься; оскільки
# колір зафарбовування не задано, він збігатиметься з кольором
# пензлика, тобто буде чорним
for i in range(6):
# створюємо цикл для рисування сторони шестикутника
    t.forward(150)          # рисуємо сторону
    t.left(60)
# повертаємо черепаху в напрямі наступної сторони
t.end_fill()                # зафарбовуємо шестикутник
# Готуємо черепаху для рисування кола
t.goto(75,0)  # пересуваємо черепаху в крайню праву точку кола
t.setheading(90)            # скеровуємо черепаху вверх
t.fillcolor('white')        # колір зафарбовування – білий
# Рисуємо коло
t.begin_fill()
t.circle(75)
t.end_fill()
t.hideturtle()              # приховуємо черепаху
```

11.1.2. Візуалізація даних засобами matplotlib

`matplotlib` – інтегрований з Python стороній пакет математичної графіки, який зазвичай використовують для візуалізації даних у вигляді графіків і діаграм. Не входячи до складу стандартної бібліотеки, він потребує додаткового встановлення, після чого стає доступним через інструкцію імпорту. Для встановлення пакета `matplotlib`, як і більшості інших сторонніх пакетів, зазвичай використовується менеджер пакетів pip3, який запускається з командного рядка операційної системи командою

```
pip3 install matplotlib
```

Дані для побудови об'єкта математичної графіки зберігаються у списках однакової розмірності, а вид об'єкта задається відповідним методом, зокрема:

```
bar()          # гістограма
hist()         # гістограма (на основі незгрупованих даних)
plot()         # графік у декартових координатах
polar()        # графік у полярних координатах
scatter()      # діаграма розсіювання (точкова діаграма)
```

Наприклад, результатом виконання наступного фрагмента коду буде графік функції `y = x sin(100x)` на проміжку `[0; 1]`.

```
# імпорт графічного модуля pyplot з пакета matplotlib
from matplotlib import pyplot as plt

# імпорт функції для обчислення синуса з модуля math
# стандартної бібліотеки
from math import sin

# генерування абсцис точок, за якими будуватиметься графік,
# тобто списку [0.0, 0.001, 0.002, 0,003, ..., 0.999, 1.0]
x = [i/1000 for i in range(1001)]
```

```
# генерування за списком абсцис списку ординат
y = [i*sin(100*i) for i in x]

plt.plot(x, y) # створення графічного об'єкта
plt.show()             # відображення графічного об'єкта
```

Графік будується в окремому вікні, яке містить панель інструментів для зміни масштабу, задання відступів і збереження рисунка в окремому файлі (рис. 11.2). Зберегти рисунок можна і автоматично, викликаши метод `savefig()` та передавши йому ім'я файлу, наприклад,

```
plt.savefig('myplot.png')
```

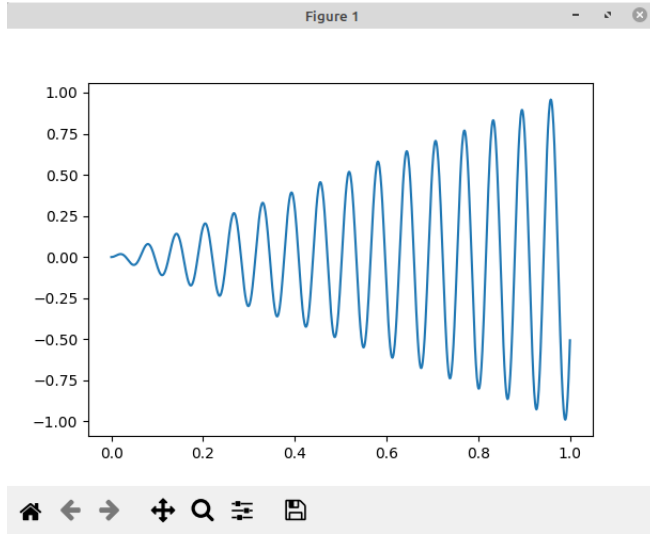


Рис. 11.2. Вікно з графічним об'єктом matplotlib

Додаткові виклики методу `plot()` з іншими аргументами-даними дають змогу отримати кілька графіків на одному рисунку.

Методи `plot()`, `bar()` тощо можуть також приймати притаманні їм іменовані аргументи. Наприклад, для використаного вище методу `plot()` це `ls` (визначає стиль лінії: `'--'` – штрихова, `':'` – пунктирна, `'-.'` –

штрихпунктирна), c (колір лінії: `'b'` – блакитний, `'g'` – зелений і т. д.), `lw` (товщина лінії у пунктах) та ін.

Крім того, графічні об'єкти мають методи форматування, серед яких:

```
plt.xlabel(s)  # додає підпис (рядок s) до осі абсцис
plt.ylabel(s)  # додає підпис (рядок s) до осі ординат
plt.legend()
# додає легенду (підпис до графіку), текст якої задається
# іменованим аргументом label методу plot()
plt.xlim(minX, maxX)
# межі системи координат по осі абсцис
plt.ylim(minY, maxY)
# межі системи координат по осі ординат
plt.xticks(<послідовність>)
# поділки осі абсцисс (аргументом можна передавати список або
# результат функції range())
plt.yticks(<послідовність>)
# поділки осі ординат (аналогічно до xticks)
```

11.2. Графічний інтерфейс користувача

В усіх попередніх розділах програміст/користувач взаємодіяв з програмою через термінал, тобто передавав програмі вхідні дані й отримував результат (за винятком попереднього розділу) через текстове вікно командної оболонки. Реальні програмні продукти зазвичай взаємодіють з користувачами через віконні графічні інтерфейси користувача (GUI[77]), маніпулюючи інформацією, використовуючи меню, кнопки, піктограми, перемикачі й інші елементи вікна (віджети).

На відміну від консольних, графічні інтерфейси, крім візуалізації, дають змогу вводити вхідну інформацію у довільному порядку та виконувати певні операції після натискання відповідних кнопок

77 Graphical User Interface.

(комбінацій клавіш) чи вибору відповідних команд з меню. Крім того, вони позбавляють необхідності перезапускати програму і повторно вводити вхідні дані, коли треба отримати результат на різних наборах даних.

Для розробки графічних інтерфейсів Python надає стандартну бібліотеку `tkinter`, яка містить готові класи для вікон і віджетів[78]. Принципи програмування деяких елементів графічного інтерфейсу з використанням цієї бібліотеки розглянуті нижче на прикладі програми для обчислення площі круга та довжини кола зазначеного радіуса, вікно якої зображено на рис. 11.3.

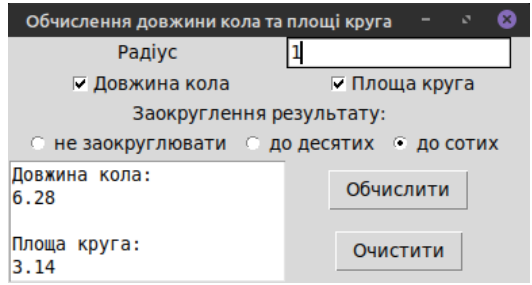


Рис. 11.3. Приклад програми з графічним інтерфейсом

11.2.1. Кореневе вікно

Для створення кореневого вікна, в якому розміщуватимуться віджети, використовується клас `Tk`. Цей клас має методи для задання певних атрибутів вікна (наприклад, заголовку, початкових розмірів, вигляду керуючих кнопок[79]), і метод `mainloop()`, призначенням якого є постійне відображення вікна на екрані та його зміна за умови настання певних подій (введення інформації, натискання кнопки, наведення вказівника миші тощо). Зовнішній вигляд вікна відповідає вікнам операційної системи, в якій виконується програма.

78 Також існують сторонні модулі для розробки GUI: wxPython, PyQt та ін.

79 У цьому посібнику розглянуто тільки основні атрибути вікон і віджетів. Атрибути, які пов'язані з вирівнюванням, розмірами, кольорами, обрамленнями й іншими декораційними елементами, не розглядаються.

Наступний код створює порожнє вікно програми для обчислення довжини кола та площі круга, яке зображене на рис. 11.4:

```
from tkinter import *
root = Tk()
root.title('Обчислення довжини кола та площі круга')
root.mainloop()
```

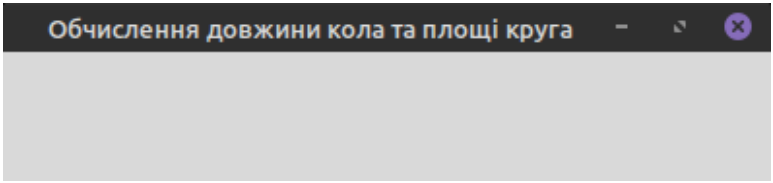


Рис. 11.4. Кореневе вікно

Будь-який віджет, який розміщуватиметься у вікні, незалежно від своєї форми, займає прямокутну область. Тому розташування (компонування) віджетів у вікні зводиться до поділу вікна на прямокутники, по одному для кожного віджета. Бібліотека `tkinter` пропонує два методи компонування віджетів – `grid()` і `pack()`. Перший з них полягає в умовному поділі вікна на сітку з пронумерованими (починаючи з 0) рядками та стовпчиками, другий – у поступовому впорядкованому заповненні вільного простору вікна віджетами, зазначаючи місця їхнього прикріплення. У цьому посібнику розглянуто метод компонування `grid()`, тобто місце розташування віджета у вікні задається номерами рядка і стовпчика.

11.2.2. Написи

Напис у вікні графічного інтерфейсу створюється за допомогою класу `Label`. Першим аргументом зазначається, в якому контейнері має бути розміщений напис[80] (у нашому випадку це вікно `root`), а іменованим аргументом `text` передається саме текст напису. Для розміщення напису

80 Це стосується не тільки напису, а й будь-якого іншого віджета.

в конкретному місці вікна аргументами `row` і `column` методу `grid()` передаються номери рядка та стовпця[81]:

```
radiusLabel = Label(root, text='Радіус')
radiusLabel.grid(row=0, column=0)
```

У підсумку матимемо вікно з написом, зображене на рис. 11.5.

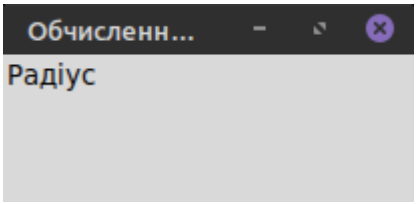


Рис. 11.5. Вікно з написом

Якщо наперед відомо, що напис ніколи не змінюватиметься впродовж виконання програми, то допускається запис коду в один рядок[82]:

```
Label(root, text='Радіус').grid(row=0, column=0)
```

Умова незмінюваності віджета впродовж виконання програми накладається тому, що після такого однорядкового запису коду доступ до об'єкта, а отже, і до його атрибутів буде втрачено, навіть якщо запис виглядатиме так:

```
radiusLabel = Label(root, text='Радіус').grid(row=0, column=0)
```

11.2.3. Поля для введення

У простих графічних інтерфейсах введення текстової інформації зазвичай відбувається в однорядкове поле для введення, яке створюється за допомогою класу `Entry` (див. рис. 11.6):

---

81 Код для створення і розташування напису та розглянутих далі інших віджетів, розміщується між інструкціями root = Tk() іtaroot.mainloop().

82 Це також стосується інших віджетів.

```
radiusEntry = Entry(root)
radiusEntry.grid(row=0,column=1)
```

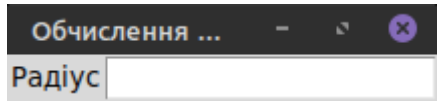


Рис. 11.6. Вікно з доданим полем для введення текстової інформації

У цьому випадку у змінній `radiusEntry` не міститиметься введене значення радіусу. Для його отримання потрібно скористатися методом `get()`:

```
radiusEntry.get()
```

11.2.4. Прапорці

Прапорці дають змогу реалізувати інтерфейс для вибору кількох варіантів. У нашій програмі вони дають змогу обчислювати довжину кола, площу круга або обидві величини. Кожен з прапорців є окремим віджетом `Checkbutton`, який асоціює свій стан з певним значенням прив'язаної до нього змінної, ім'я якої передається іменованим аргументом `var`, а значення – іменованим аргументом `value`. Іменований аргумент `text` визначає мітку (напис біля прапорця).

Додати два прапорці опцій з мітками "Довжина кола" і "Площа круга" (див. рис. 11.7) можна, наприклад, так:

```
calcL = IntVar()
calcS = IntVar()
Checkbutton(root, text='Довжина кола', var=calcL)\
           .grid(row=1, column=0)
Checkbutton(root, text='Площа круга', var=calcS)\
           .grid(row=1, column=1)
```

Кожен з прапорців асоціюється зі своєю змінною описаного в бібліотеці `tkinter` типу `IntVar`. Якщо прапорець активувати, то всередині квадратика з'явиться галочка, а значення відповідної змінної становитиме 1. Значення змінної, асоційованої з неактивованим

прапорцем, дорівнює 0. Для отримання поточного стану цих значень використовується метод `get()`, наприклад,

```
calcS.get()
```

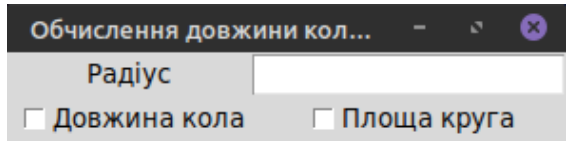


Рис. 11.7. Вікно з доданими прапорцями опцій

Значення аргументів методу `grid()` вказують, що прапорці розміщуються в рядку номер 1 і стовпчиках під номерами 0 та 1, відповідно.

11.2.5. Розміщення віджета у кількох рядках/стовпчиках

Для того, щоб розмістити віджет більше ніж у одному рядку чи стовпчику, методу `grid()` передаються параметри `rowspan` і `columnspan`, які вказують, на скільки рядків чи стовпців, відповідно, потрібно розтягнути віджет. У нашому графічному інтерфейсі напис “Заокруглення результату:” простягається на два стовпці (див. рис. 11.3, 11.8). Цього можна досягти так:

```
Label(root, text='Заокруглення результату:')\
     .grid(row=2, column=0, columnspan=2)
```

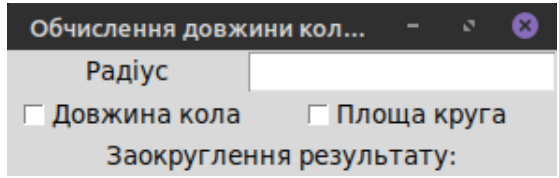


Рис. 11.8. Вікно з віджетом-написом завширшки два стовпчики

11.2.6. Контейнери

З рис. 11.3 видно, що способи заокруглення відображені у трьох стовпчиках, а основна сітка кореневого вікна налічує два. Для вирішення цієї невідповідності призначені контейнери (тип `Frame`). Контейнер можна розглядати як звичайне вікно, розташоване всередині іншого вікна (у нашому випадку кореневого). Всередині контейнера можна утворювати нову сітку та компонувати віджети уже в ній, передаючи першим параметром ім'я цього контейнера). Створимо в наступному рядку сітки кореневого вікна контейнер і розмістимо його на два стовпці (тобто на всю ширину інтерфейсу) вшир. Оскільки контейнер надалі використовуватиметься, то використовувати однорядковий запис коду не можна, щоб не втратити доступ до контейнера:

```
rounding = Frame(root)
rounding.grid(row=3, column=0, columnspan=2)
```

11.2.7. Перемикачі

Перемикачі дають змогу реалізувати інтерфейс для вибору одного з кількох варіантів. У нашій програмі вони дають змогу або не заокруглювати обчислені значення довжини кола та площі круга, або заокруглювати до десятих, або до сотих. Кожен з перемикачів у групі є окремим віджетом `Radiobutton`, але всі вони асоціюються з тією самою змінною, ім'я якої передається іменованим аргументом `var`, а значення – іменованим аргументом `value`. Іменований аргумент `text` визначає мітку (напис біля перемикача).

Додати три перемикачі з мітками "не заокруглювати", "до десятих" і "до сотих" (див. рис. 11.9) можна, наприклад, так:

```
roundTo = StringVar()
rb1 = Radiobutton(rounding, var=roundTo,
                  value='без заокруглення')
rb1['text'] = 'не заокруглювати'
rb1.grid(row=0, column=0)
rb2 = Radiobutton(rounding, var=roundTo, value='1')
```

```
rb2['text'] = 'до десятих'
rb2.grid(row=0, column=1)
rb3 = Radiobutton(rounding, var=roundTo, value='2')
rb3['text'] = 'до сотих'
rb3.grid(row=0, column=2)
roundTo.set('без заокруглення')
```

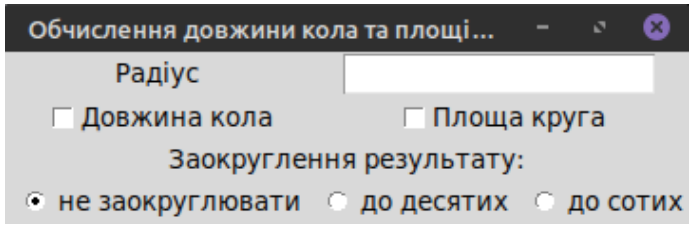


Рис. 11.9. Вікно з доданою групою перемикачів

Як бачимо, першим параметром, який передається перемикачеві, є не кореневе вікно, а контейнер, створений у попередньому підрозділі.

Кожен з трьох перемикачів асоціюється зі змінною `roundTo` типу `StringVar`[83] (описаний в бібліотеці `tkinter` тип, аналогічний до типу `str`, але зі своїми методами). Активація перемикача присвоює цій змінній відповідне цьому перемикачу значення `value` (для першого перемикача це `'без заокруглення'`, для другого та третього – кількість десяткових знаків, які будуть використовуватися під час заокруглення вбудованою функцією `round()`, але у форматі рядків (через тип `StringVar`): `'1'` та `'2'`, відповідно).

Для демонстрації іншого способу задання атрибутів віджета, атрибут `text` заданий у окремому рядку.

Методом `set()` у останньому рядку коду задається значення перемикача за замовчуванням.

83 Тип StringVar обрано для різноманітності, можна було також використовувати тип IntVar, задавши відповідні цілочислові значення для кожного перемикача.

11.2.8. Кнопки

Кнопка (клас `Button`) – віджет, натискання якого викликає функцію, ім'я якої передається іменованим аргументом `command`. Графічний інтерфейс нашої програми містить кнопки "Обчислити" й "Очистити" (див. рис. 6). Натискання першої викликатиме виконання обчислень і запис результату у відповідне поле (буде створене в наступному підрозділі), а друга – очищення полів з вхідними даними (радіус) і результом (довжину кола та площу круга) для забезпечення можливості нових обчислень без повторного запуску програми.

Створення кнопок нічим не відрізняється від створення розглянутих вище віджетів (результат зображений на рис. 11.10):

```
Button(root, text='Обчислити', command=calculate)\
      .grid(row=4, column=1)
Button(root, text='Очистити', command=clear)\
      .grid(row=5, column=1)
```

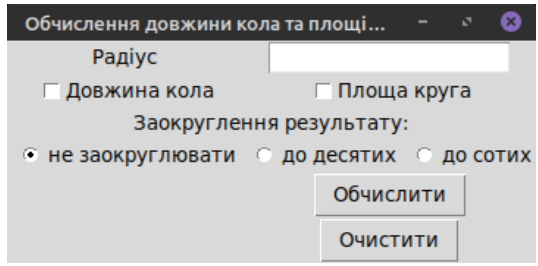


Рис. 11.10. Вікно з доданими кнопками

Щоб програма не видавала помилки, потрібно описати асоційовані з цими кнопками функції `calculate()` і `clear()`. Зазвичай вони описуються в іншому файлі (тоді потрібен імпорт) або в поточному файлі одразу після інструкцій імпорту:

```
def calculate():
       pass
def clear():
       pass
```

Тіла цих функцій будуть наведені нижче, в підрозділі “Обробка подій”, бо в них використовуватиметься наразі не створене поле для результату. Тому на цьому етапі просто прописується порожня інструкція `pass`, яка дає змогу коректно запустити програму. Іншими словами, поточний код означає, що натискаючи на кнопки не відбуватиметься нічого, але сам процес натискання дозволений.

11.2.9. Текстові поля

Текстове поле (віджет `Text`) у найпростішому випадку використовується для роботи з багаторядковими рядками, тобто значеннями типу str, які містять символи кінця рядка `'\n'`. У цьому випадку індексуються не просто порядкові номери символів, а номери рядків (починаються з 1) та номери стовпців (починаються з 0).

Для простих операцій з текстовим полем (екземпляром класу `Text`) використовують методи `insert()`, `delete()` та `get()`[84].

Методу `insert()` першим параметром передається позиція, в яку має вставитись рядок, переданий у другому параметрі. Наприклад, код

```
someText = Text()
someText.insert('1.2', 'Деякий текст')
```

означає, що текст `'Деякий текст'` буде вставлений, починаючи з позиції “перший рядок, третій стовпчик”.

Для позначення позицій також використовуються символічні імена. Наприклад, для вилучення всього тексту з текстового поля `someText` методу `delete()` можна передати початкову позицію у формі `'<номер рядка>.<номер стовпчика>'`, а кінцеву у вигляді символьного імені `END`:

```
someText.delete('1.0', END)
```

Метод `get()` використовують для отримання частини тексту між початковою та кінцевою позиціями, наприклад,

---

84 Ці самі методи доступні для віджета Entry, тільки індексація там одновимірна, як для типу str (задається цілим числом – порядковим номером символу у рядку).

```
someText.get('1.2', '2.1')
```

Створимо на основі класу Text текстове поле для відображення результатів обчислення (див. рис. 11.11):

```
result = Text(root, width=25, height = 5)
result.grid(row=4, column=0, rowspan=2)
```

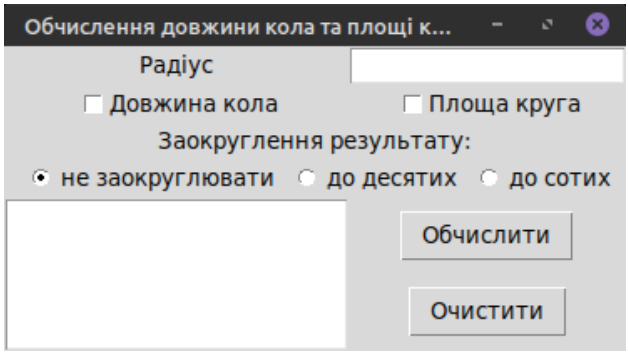


Рис. 11.11. Вікно з усіма елементами графічного інтерфейсу

Параметрами `width` і `height` задано точні розміри поля (5 рядків по 25 символів у кожному), бо за замовчуванням вони значно більші. Код для запису результату в це поле буде сформований у наступному підрозділі.

11.2.10. Опрацювання подій

Бібліотека `tkinter` дає змогу опрацьовувати велику кількість подій, прив'язаних до різноманітних змін у графічному інтерфейсі: наведення вказівника миші, клацання певною кнопкою миші на об'єкті, натискання клавіші на клавіатурі, зміна поля, прапорця чи перемикача тощо.

У рамках нашої програми опрацювання подій стосується виклику функції `clear()` у разі натискання кнопки (клацанні лівою кнопкою миші на кнопці) "Очистити" та функції `calculate()` у разі натискання кнопки "Обчислити". Вище, під час створення кнопок, ми прив'язали ці функції до відповідних їм кнопок за допомогою іменованого аргументу command.

Тепер наведемо код тіл цих функцій з коментарями (використання числа "пі" у тілі функції `calculate()` потребує додаткового імпорту, тобто інструкції `from math import pi`):

```
def calculate():
    # отримуємо текстове значення радіуса з поля для введення
    # та перетворюємо його в дійсне число
    r = float(radiusEntry.get())
    # отримуємо значення асоційованої з перемикачами змінної,
    # щоб знати, як виконувати заокруглення
    rb = roundTo.get()
    # створюємо порожній рядок для формування тексту результату
    text = ''
    # для зменшення обсягу коду обчислимо заздалегідь
    # довжину кола і площу круга
    l = 2*pi*r               # довжина кола
    s = pi*r**2              # площа круга
    if rb != 'без заокруглення':
    # якщо потрібно заокруглювати, то заокруглюємо до
    # кількості десяткових знаків, перетворених в ціле число:
        l = round(l, int(rb))
        s = round(s, int(rb))
    if calcL.get():
        # якщо активований прапорець "Довжина кола", то
        # додаємо текст результату
        text += f'Довжина кола:\n{l}\n\n'
    if calcS.get():
        # якщо активований прапорець "Площа круга", то
        # додаємо текст результату
        text += f'Площа круга:\n{s}'
    result.insert('1.0', text)
    # вставляємо результат у текстове поле
```

Важливо! Якщо не використовувати метод `get()`, тобто замість інструкції `if calcL.get():` вживати інструкцію `if calcL:`, то програма

не видасть синтаксичної помилки чи помилки виконання, але умова виконуватиметься завжди, бо об'єкт з іменем `calcL` існує. За таких обставин матимемо семантичну помилку, яка призведе до некоректної роботи програми (незалежно від активованих прапорців завжди обчислюватиметься і довжина кола, і площа круга).

```
def clear():
    # очищаємо поле для введення радіуса
    radiusEntry.delete(0, END)
    # очищаємо поле з результатом
    result.delete('1.0', END)
```

11.2.11. Стандартні діалоги

Реальні програми містять стандартні діалоги: вибір файлу, повідомлення про помилки тощо. Усі вони блокують головне вікно програми та повертають керування тільки після закриття вікна діалогу. Бібліотека tkinter пропонує набір типових стандартних діалогів. Наприклад, модуль messagebox містить функції для реалізації вікон таких діалогів, серед яких: `askyesno()`, `showwarning()`, `showinfo()`. Перша з них створює вікно з текстом і двома кнопками типу "ОК" та "Відміна" (написи на кнопках відрізняються залежно від операційної системи, під якою працює програма). Дві наступні відображають вікно з текстовим повідомленням і відрізняються лише піктограмами біля тексту. Першим параметром цим функціям передається заголовок вікна, а другим – саме текст повідомлення.

Реалізуємо в нашій програмі обробку винятків, яка в разі винятку відображатиме вікно стандартного діалогу, а саме: якщо в поле для введення радіусу буде введене не число, то після натискання на кнопку "Обчислити" програма виводитиме на екран вікно з відповідним повідомленням-попередженням (див. рис. 11.12). Для цього імпортуємо функцію `showwarning()` з модуля `messagebox` бібліотеки `tkinter`:

```
from tkinter.messagebox import showwarning
```

Після цього у функції `calculate()` рядок, у якому введене в поле для введення значення радіуса перетворюється в дійсне число переносимо в блок `try`, а у блоці `except` прописуємо відображення діалогового вікна “Некоректні дані” з текстом “Радіус повинен бути числом” та завершуємо виконання функції:

```
try:
    r = float(radiusEntry.get())
except:
    showwarning('Некоректні дані', 'Радіус має бути числом')
    return None
```

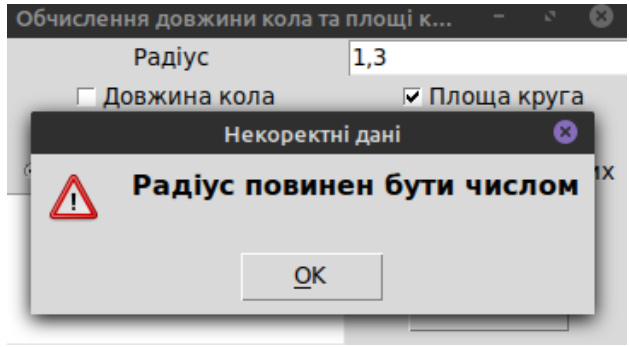


Рис. 11.12. Вікно стандартного діалогу

Отже, остаточний код програми для обчислення довжини кола та площі круга за заданим радіусом набуде вигляду (усі коментарі опущені):

```
from tkinter import *
from tkinter.messagebox import showwarning
from math import pi

def calculate():
    try:
        r = float(radiusEntry.get())
    except:
```

```
        showwarning('Некоректні дані', 'Радіус повинен бути\
                    числом')
        return None

    rb = roundTo.get()
    text = ''
    l = 2*pi*r
    s = pi*r**2

    if rb != 'без заокруглення':
        l = round(l, int(rb))
        s = round(s, int(rb))

    if calcL.get():
        text += f'Довжина кола:\n{l}\n\n'

    if calcS.get():
        text += f'Площа круга:\n{s}'

    result.insert('1.0', text)

def clear():
       radiusEntry.delete(0, END)
       result.delete('1.0', END)

root = Tk()
root.title('Обчислення довжини кола та площі круга')
Label(root, text='Радіус').grid(row=0, column=0)
radiusEntry = Entry(root)
radiusEntry.grid(row=0,column=1)

calcL = IntVar()
calcS = IntVar()
```

```
Checkbutton(root, text='Довжина кола', var=calcL)\
           .grid(row=1, column=0)
Checkbutton(root, text='Площа круга', var=calcS)\
           .grid(row=1, column=1)

Label(root, text='Заокруглення результату:')\
     .grid(row=2, column=0, columnspan=2)

rounding = Frame(root)
rounding.grid(row=3, column=0, columnspan=2)

roundTo = StringVar()
rb1 = Radiobutton(rounding, var=roundTo,
                  value='без заокруглення')
rb1['text'] = 'не заокруглювати'
rb1.grid(row=0, column=0)
rb2 = Radiobutton(rounding, var=roundTo, value='1')
rb2['text'] = 'до десятих'
rb2.grid(row=0, column=1)
rb3 = Radiobutton(rounding, var=roundTo, value='2')
rb3['text'] = 'до сотих'
rb3.grid(row=0, column=2)
roundTo.set('без заокруглення')

Button(root, text='Обчислити', command=calculate)\
      .grid(row=4, column=1)
Button(root, text='Очистити', command=clear)\
      .grid(row=5, column=1)

result = Text(root, width=25, height = 5)
result.grid(row=4, column=0, rowspan=2)

root.mainloop()
```

Питання та завдання для самоперевірки

1. Використавши власноруч створену функцію для рисування правильного трикутника, аргументами якої є сторона трикутника та колір, написати програму для рисування трьох трикутників різного розміру і кольору:

2. Використовуючи пакет matplotlib, візуалізувати дані про розподіл оцінок у вигляді гістограми.

Бали	0-50	51-70	71-89	90-100
Кількість студентів	3	9	5	2

3. Якщо не використовувати кнопку “Очистити”, а вручну стерти значення радіуса з поля для введення і ввести нове значення, то після натискання на кнопку “Обчислити” результат нового обчислення буде доданий перед результатами попереднього, якщо їх теж не витерти вручну. Змінити код програми так, щоб передбачити цю ситуацію, тобто, щоб нові результати не змішувалися з попередніми ні за яких обставин.

4. Подібно до того, як передбачено ситуацію некоректного введення радіуса, змінити код програми так, щоб він опрацьовував ситуацію, коли жоден з прапорців не активований.

ВІДПОВІДІ ДО ПИТАНЬ І ЗАВДАНЬ ДЛЯ САМОПЕРЕВІРКИ

Відповіді до розділу 1

1. Порівняно невисока швидкість виконання програми (продуктивність).

2. `100. Hi.`
   Ввести `sto` і натиснути *Enter*.

3. Оскільки синтаксис Python не порушено, але значення знаменника (виразу у дужках) дорівнює 0, то з'явиться повідомлення про помилку виконання: `ZeroDivisionError: division by zero` (ділення на нуль).

4. Частини, які можуть бути змінені, виокремлені жирним шрифтом:

   ```
   from math import sqrt
   data = 100
   result = sqrt(data)
   print('Результат: ', result)
   ```

   Якщо Ви не розумієте, чому саме так, поверніться до цього питання після ознайомлення з відповідними розділами.

Відповіді до розділу 2

1. Ні. Попередній рядок може належати тому самому блоку інструкцій, що і розглядуваний, а рядок з двокрапкою (перший рядок складеної інструкції) може бути вище.

2. Тільки `Day_1  i ro`.
   `1st_day` не можна, бо ім'я не може починатися цифрою;
   `or` є зарезервованим (ключовим) словом Python;
   `day-1` використовує заборонений в іменах змінних символ "мінус".

3. `some_func(value)`

4. `value.some_method()`

5. У заданому коді пропущена функція `print()`:
   ```
   a = 1; b = 2; c = 3
   print((a + c) * b)
   ```

6. Порожній рядок.

7. Так, використавши аргумент `file` (приклад наведений у відповідному розділі).

8. `print('2x =', 2*float(input('x = ')))`

Відповіді до розділу 3

1. `x ** 0.5`
   (квадратний корінь рівносильний степені 1/2).

2. ```
   c = 3+4j              # комплексне число
   cc = c.conjugate()    # комплексно спряжене число
   abs2 = abs(c)**2      # квадрат модуля комплексного
                         # числа
   prod = c * cc         # добуток числа на комплексно
                         # спряжене
   ```
   ```

```
print(abs2, prod)            # 25.0 (25+0j)
print(abs2 == prod)          # True
```

3.
```
# вводимо дані
hours = int(input('Години: '))
minutes = int(input('Хвилини: '))
seconds = int(input('Секунди: '))
# обчислюємо кількість секунд від початку доби
totalSeconds = hours*3600 + minutes*60 + seconds
# додаємо n секунд
totalSeconds += int(input('n = '))
# відкидаємо "зайві" секунди, якщо їхня кількість
перевищила добу,
# взявши остачу від ділення на кількість секунд у добі
totalSeconds = totalSeconds % 86400
# обчислюємо новий час
hours = totalSeconds // 3600
minutes = (totalSeconds % 3600) // 60
seconds = (totalSeconds % 3600) % 60
# виводимо результат
print('Години:', hours, '; Хвилини:', minutes, ';
Секунди:', seconds)
```

4. `1320`.
Усі інші числа є різними поданнями одного й того самого дійсного числа `132.0`

5. Правильна послідовність дій:
1) піднесення 3 до квадрата: `3**2` (значення `9` типу `int`);
2) ділення значення, отриманого на попередньому кроці, на `7` (значення виразу 9/7 дорівнює `1.2857142857142858` типу `float`);
3) додавання двійки до значення, отриманого на

попередньому кроці (значення виразу 2+1.2857142857142858 дорівнює 3.2857142857142858 типу float);
4) заокруглення отриманого на попередньому кроці значення до одного значення після коми (результат виконання функції round(3.2857142857142858, 1) дорівнює 3.3 типу float);
5) "обтинання" дробової частини отриманого на попередньому кроці значення (результат виконання функції int(3.3) дорівнює 3 типу int);
6) обчислення остачі від ділення числа 200 на значення виразу з попереднього кроку (значення виразу 200%3 дорівнює 2 типу int);
7) віднімання від одиниці значення виразу з попереднього кроку (значення виразу 1.-2 дорівнює -1.0 типу float).
Отже, значенням початкового виразу є число -1 типу float, тобто -1.0.

6. Метод is_integer() не перевіряє тип числа, а лише те, чи значення цього дійсного числа дорівнює якомусь цілому числу, чи ні. Оскільки дійсне число 2.0 дорівнює цілому числу 2, то застосування цього методу до змінної a дасть True. Відповідно, застосування цього методу до змінної b дасть False.

Відповіді до розділу 4

1. Створити текстовий файл з кодом на мові Python. Цей файл автоматично перетвориться в об'єкт модуля після виконання інструкції import.

2.
```
# перший спосіб
import data
number = 100
print(number * data.number)
# другий спосіб
from data import number as numberFromData
number = 100
print(number * numberFromData)
```

3.
```
from time import sleep
print(help(sleep))
```
Призупиняє виконання програми на задану кількість секунд.

4. Спочатку інтерпретатор обчислює значення `math.pi`: 3.141592653589793. Потім `math.degrees(3.141592653589793): 180.0`.
Далі `math.exp(180.0): e180.0 = 1.4893842007818383e+78`
Далі `log(1.4893842007818383e+78): ln(e180.0) = 180.0`
Функція `int(180.0)` остаточно поверне ціле число `180`.
Код для перевірки:
```
import math
print(int(math.log(math.exp(math.degrees(math.pi)))))
```

5. Це пояснюється тим, що в модулі `math` є атрибут з іменем `e`, значення якого дорівнює експоненті. Оскільки інструкція `from math import *`, яка імпортує всі імена з модуля `math`, знаходиться після інструкції присвоєння `e = 3`, то значення **3** буде втрачене і одиниця відніматиметься вже від значення експоненти `2.718281828459045`. Цього б не трапилось, якби інструкція імпорту була першим

рядком коду, але тоді стало б недоступним значення експоненти. Тому найкращим варіантом вважається використання інструкції `import math`, яка дасть змогу використовувати як значення `e = 3` та значення експоненти `math.e`.

7.
```
import random as r
r.uniform(0, 100)
```

Відповіді до розділу 5

1. `a == 3 or b > 5` або `not(a != 3 and b <= 5)`

2. У логічному виразі `greeting = 'Слава Україні!'` замість оператора порівняння `==` використано оператор присвоєння `=`.

3.
```
n = input('Введіть ціле число')
if n:
    if n > 0:
        print('Ви ввели додатне число')
    else:
        print("Ви ввели від'ємне число")
else:
    print('Введене вами число – нуль')
```

4. `lambda` є зарезервованим словом Python і не може використовуватися як ім’я змінної.

5. 6.

Відбудеться перехід у тіло циклу, оскільки значення змінної 3 є більшим за 0. В тілі циклу значення змінної збільшиться вдвічі, а інструкція break змусить інтерпретатор одразу покинути цикл.

6. Оскільки `range(0, 2, 10)` згенерує числа від 0 до 10-1=9 з кроком 2, тобто 0, 2, 4, 6, 8, то кількість ітерацій циклу дорівнюватиме 5. Символ підкреслення _ є нічим іншим, як іменем змінної, яка почергово прийматиме згенеровані числові значення.

7. Перший цикл: `1010101010`
   Другий цикл: `9876543210`
   Третій, 4-1й та п'ятий цикли виконуватимуться нескінченно.

8.
```
a = float(input())
d = float(input())
n = int(input(()))
for i in range(n-1):
    a += d
print(a)
```
Програма обчислює n-й член арифметичної прогресії. Обидва цикли можна замінити виразом `a += (n-1)*d`.

Відповіді до розділу 6

1. `YES`
   `O`
   `yes`
   `o`

Рядок `s` складається з п'яти (не шести!) символів: літер `'y'`, `'e'` і `'s'`, символу нового рядка `'\n'` і літери **`'o'`**. Тобто, після застосування методу `upper()`, який переводить усі символи-літери у верхній регістр, першій функції `print()` буде передано рядок `'YES\nO'`. Оскільки рядки є незмінюваним типом, то застосування методу `upper()` не вплине на значення змінної `s`, що засвідчує результат виконання другої функції `print()`.

2. Головна ідея полягає в перезаписуванні значення змінної `s`, наприклад,
```
s = s[:2] + s[3:]
```

3.
```
s = 'abcde'
print(s[::-1]) # edcba
```

4. Для розуміння коду ознайомитись з таблицею ASCII.

```
# Знаходимо відстань між кодами символів 'a' і 'A'
# (можна будь-яких інших літер в різних регістрах)
distance = ord('a') – ord('A')
# Просимо користувача ввести будь-яку велику латинську
літеру
bigChar = input('Введіть велику латинську літеру: ')
# Додаємо до коду введеної літери відстань, отримавши
код шуканої малої
# літери, і відтворюємо символ за цим кодом
smallChar = chr(ord(bigChar) + distance)
# Виводимо результат на екран
print(smallChar)
```

5.
```
text = input('Введіть текст:\n')
counter = 0    # обнулюємо лічильник символів-цифр
```

```
    for symb in text:            # обходимо текст
    посимвольно
        if symb.isdigit():
            # якщо поточний символ – цифра, то збільшуємо
            # лічильник на 1
            counter += 1
print('Кількість цифр у тексті:', counter)
```

6. `'r'`

7.
```
    with open('squares.txt', 'a') as f:
        for i in range(10):
        # цикл по рядках (i = 0, 1, 2, ..., 9)
            for j in range(1,11):
                # цикл по стовпчиках (j = 1, 2, ..., 10)
                '''На перетині i-го рядка та j-го стовпчика
                   записується квадрат числа, кількість
                   десятків якого дорівнює номеру рядка, а
                   кількість одиниць – номеру стовпчика.
                   Крім того, відокремлюємо числа знаком
                   табуляції'''
                f.write(str((10*i+j)**2)+'\t')
            f.write('\n') # переходимо на новий рядок у
    файлі
```

8.
```
    with open('data.txt') as f:
        profit = 0 # початкове значення річного прибутку – 0
        for line in f:           # цикл по рядках файлу
            '''Знаходимо індекс (порядковий номер)
               послідовності ': ', за якою йде місячний
               прибуток'''
            index = line.find(': ')
            '''Додаємо вийнятий за допомогою зрізу та
               перетворений на дійсне число місячний
```

```
прибуток до загального'''
        profit += float(line[index+2:])
print(f'Річний прибуток: {profit}')
```

9.
```
1) 0
2) runtime error
3) 0
4) runtime error
5) 02
6) 00
7) 2
8) runtime error
9) syntax error
10) runtime error
11) runtime error
12) m
```

Відповіді до розділу 7

1. `[3, 4, 5] 1`

   Спочатку буде виведено зріз списку (елементи, починаючи з індексу 2 (третій елемент) і закінчуючи передостаннім (оскільки останній елемент з індексом -1 не входить у зріз). Після цього через пробіл буде виведено перший елемент (з індексом 0). Позаяк зріз не змінює списку, до якого він застосовується, то це буде 1.

2. `скло`

   У першому рядку змінній `l` присвоюється список, утворений з літер рядка `'скол'`, тобто `['с', 'к', 'о', 'л']`. У другому рядку спочатку поелементно формується список з двох елементів (у квадратних дужках) шляхом виконання двох однакових операцій вилучення елемента

з кінця списку l. У підсумку список l матиме вигляд ['с', 'к'], а методу extend() передасться список, який він долучить до l. У третьому рядку функції print() передається рядок, утворений методом join() зі списку l (який має тепер вигляд ['с', 'к', 'л', 'о']), тобто на екрані відобразиться слово скло.

3. Під час перевірки на входження переводимо всі літери у нижній (можна верхній) регістр методом lower(), крім того, перевіряємо методом isalpha(), чи є символ літерою. Для забезпечення виконання обох умов використовуємо логічний оператор and. Приклад коду:

```
s = 'Київ – столиця України'
uniqueSymbols = []
for symbol in s:
    if symbol.lower() not in uniqueSymbols \
    and symbol.isalpha():
        uniqueSymbols.append(symbol.lower())
print(uniqueSymbols)
```

4. [[0]*3]*3
Вираз [0]*3 створює список [0, 0, 0], який потім "розмножується" тричі.

5. Ні, бо змінній x було присвоєне значення типу (рядок).

6. 12
Оскільки список є змінюваним типом, то змінні l1 і l2 вказують на той самий об'єкт списку, який після застосування методу append() набуде вигляду [1, 2, 3]. Операція l1 + l2 утворить новий список [1, 2, 3, 1, 2, 3], сума елементів якого дорівнює 12.

7. Бо кортеж, на відміну від списку, є незмінюваним типом даних.
8. Вираз-генератор, який послідовно генерує числа 1, 2, 3, не зберігаючи їх у жодну структуру. Помилково вважати, що це генератор кортежу, бо у зв'язку з незмінюваністю кортежів генераторів для них не існує.

9.
```
t = ("нуль", "один", "два", "три", "чотири",
     "п'ять", "шість", "сім", "вісім","дев'ять")
x = int(input())
for item in str(x):
    print(t[int(item)], end=' ')
```

10. Замість ініціалізації порожнього словника встановлюємо нульові початкові значення лічильників для ключів `0` і `1`:

```
from random import randint
bits = [randint(0,1) for i in range(100)]
freq = {0: 0, 1: 0}
for bit in bits:
    freq[bit] += 1
print(freq)
Або можна скористатися методом setdefault():

from random import randint
bits = [randint(0,1) for i in range(100)]
freq = {}
for bit in bits:
    freq.setdefault(bit, 0)
    freq[bit] += 1
print(freq)
```

11. `{'один': 1}`.
    Інструкція `myDict.pop(1)` вилучає елемент з ключем (а не значенням) 1.

12.
```
amounts = {}  # створюємо порожній сновник
for author, books in library.items():
    # цикл по авторах
    amounts[author] = 0
    # встановлюємо лічильник книг автора в 0
    for book, amount in books.items():
        # цикл по книгах автора
        amounts[author] += amount
        # збільшуємо лічильник на кількість примірників
print(amounts)       # друк результуючого словника
```

13. `4`
    Вираз у квадратних дужках є списком з 5 елементів: трьох чисел (одиниці і двох двійок), рядка `'1'` і кортежу `(1, 2)`. Функція `set()` утворить з цих елементів множину, видаливши дублікат числа `2` (кортеж з одиниці та двійки не зачіпатиметься, бо він інтерпретується як один об'єкт). Функція `len()` поверне кількість елементів у цій множині, тобто `4`, що і буде виведено на екран функцією `print()`.

14. `Один`.
    Метод `add()` мав би додати до множини ціле число `1` (кількість елементів у цій множині). Позаяк значення логічного виразу `1 == 1.0` дорівнює `True`, то інтерпретатор вважає, що такий елемент (одиниця) вже є у множині. Через це множина `s` залишиться незмінною.

Відповіді до розділу 8

1. Кожна. Функції, у тілі яких немає інструкції `return`, повертають значення `None`.
2. Оскільки виклик функції відбувається до її опису, то інтерпретатор видасть повідомлення про помилку
   `NameError: name 'f' is not defined`

3. Двома (див. *https://www.python.org/dev/peps/pep-0008/#blank-lines*)

4. `None`
   Метод `sort()` не повертає жодного значення, він просто відсортовує список `l`. Оскільки метод по суті є функцією, а функція, яка не має явної інструкції `return`, повертає за замовчуванням `None`, то виконання другого рядка коду запише це значення у змінну `l`. Для отримання відсортованого списку `[1, 2, 3]` другий рядок коду має набувати вигляду `l.sort()`.

5. Поліморфізм функцій полягає в тому, що, залежно від типів аргументів, функція може виконувати різні дії. Наприклад, наведена нижче функція `double()` у разі передачі їй числа виконає збільшення цього числа вдвічі (множення на `2`), а у разі передачі рядка виконає дублювання цього рядка:

6. ```
   def double(arg):
       print(arg*2)
   double(1)             # 2
   double('1')           # 11
   ```
7. `10`
   Передаючи глобальну змінну **x** (значення якої 10) функції
   ```

`make_5()`, інтерпретатор робить ім'я **x** локальним. Саме цим локальним іменем оперує функція `make_5()`: перевіряє, чи воно асоційоване зі значенням `5` і виконує операцію присвоєння за негативної відповіді. Після завершення роботи функції `make_5()` головна програма "забуває" значення локальної змінної `x` і передає функції `print()` однойменну глобальну змінну, значення якої не змінилося.

8. `Alice {'age': 18, 'job': 'student'}`
   Рядок `'Alice'` буде переданий позиційному аргументу `person`, а конструкція `**info` збере решту іменованих аргументів у словник.

9. `91 66 82 0`
   Число `91` буде передано першому аргументу `algebra`. Конструкція `*(66, 82)` розпакує числа `66` і `82` і передасть їх відповідно аргументам `geometry` і `history`. Оскільки четвертого аргументу `english` у виклику функції немає, то йому передасться значення за замовчуванням `0`.

10. `print(*s, sep='\n')`

11. Вираз `[[]]` не є порожнім списком, а позначає список, який складається з одного елемента, а саме – порожнього списку. У цьому можна переконатися, подивившись на результат виклику функції `len([[]])`, який дорівнюватиме `1`.

Відповіді до розділу 9

1. Програма аварійно завершиться з відповідним цьому винятку повідомленням про помилку.

2. Менеджер контексту `with` забезпечує закриття файлу навіть, коли у блоці інструкцій щось піде "не так". В інструкції `try` гарантувати виконання завершальних операцій, зокрема й закриття файлу, може блок `finally`. Тому найбільш наближеним до заданого коду (за умови, що файл *filename.txt* існує) є такий варіант:

```
try:
    file = open('filename.txt')
    # блок інструкцій
finally:
    file.close()
```

Відповіді до розділу 10

1. Python підтримує обидві парадигми програмування і дозволяє використовувати їх у поєднанні.

2. `MyClass` (див. *https://www.python.org/dev/peps/pep-0008/#class-names*). Стиль `my_class` рекомендовано використовувати для імен функцій, а `myClass` – змінних. Решта імен не є синтаксичними помилками, але згідно з PEP 8 не рекомендовані до використання.

3. Метод `__init__()` (конструктор класу).

4. На виклик функції.

5. Треба перевизначити його в дочірньому класі (підкласі), і всередині нього в потрібному місці викликати версію цього метода суперкласу, використавши синтаксис `<Ім'я_Суперкласу>.<ім'я_методу>(self, <аргументи>)`. Приклад наведений у завданні 7.

6.
```
# модуль (файл) vector.py

class Vector:

    def __init__(self, x, y):# конструктор класу
        self.x = x
        self.y = y

    def __mul__(self, vector): # перевантаження
оператора
        return self.x * vector.x + self.y * vector.y

# файл основної програми

from vector import Vector
a = Vector(2, -3)
b = Vector(5, 1)
print(a * b) # буде виведено 7 (2 * 5 + (-3) * 1)
```

7.
```
class Vector3D(Vector):

    def __init__(self, x, y, z):
        Vector.__init__(self, x, y)
        self.z = z
    def __mul__(self, vector):
        return Vector.__mul__(self, vector) + self.z *
vector.z
```

Відповіді до розділу 11

1.
```
import turtle          # імпортуємо модуль
t = turtle.Turtle()    # створюємо екземпляр черепахи
t.hideturtle()         # приховуємо стрілку
t.width(3)             # задаємо ширину пензлика

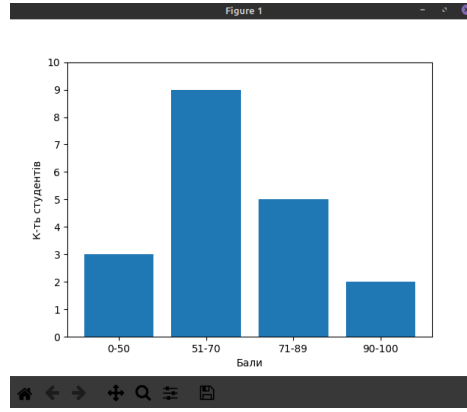
def draw_triangle(a, color):
    # функція рисування трикутника
    t.pencolor(color)        # задаємо колір лінії
    for i in range(3):       # цикл для трьох сторін
        t.forward(a)         # рисуємо сторону і
        t.left(120)          # повертаємось в напрямі
наступної

def go_to_next(a, step):
    '''Переміщує чепепаху в початкову вершину наступного
       трикутника на відстані step між сусідніми
       вершинами трикутників'''
    t.up() # піднімаємо пензлик, щоб не залишати слід
    t.forward(a + step)      # пересуваємо черепаху
    t.down()                 # опускаємо пензлик

sizes = [20, 40, 60]         # сторони трикутників
colors = ['red', 'green', 'blue']   # їхні кольори
for i in range(3):
    # організовуємо цикл для трьох трикутників
    draw_triangle(sizes[i], colors[i])
    # рисуємо трикутник
    go_to_next(sizes[i], 10)
    # і переходимо до наступного
```

2.
```
from matplotlib import pyplot as plt
# задаємо підписи стовпців гістограми
```

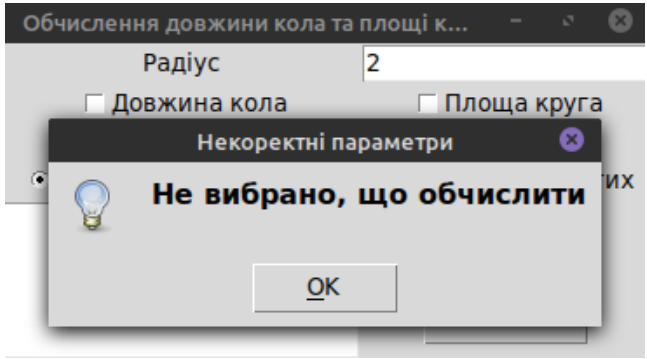
```
marks = ['0-50', '51-70', '71-89', '90-100']
studNumb = [3, 9, 5, 2]
# задаємо частоти (висоти стовпців)
plt.bar(marks, studNumb)     # створюємо гістограму
plt.xlabel('Бали')           # назва горизонтальної осі
plt.ylabel('К-ть студентів') # назва вертикальної осі
# задаємо діапазон поділок вертикальної осі, вважаючи
# висоту гістограми на одиницю більшою за максимальну
# частоту (висоту найвищого стовпця)
plt.yticks(range(0, max(studNumb)+2))
plt.show()                   # відображаємо гістограму
```



3. Для цього у тілі функції `calculate()` перед рядком
`result.insert('1.0', text)`
достатньо додати інструкцію
`result.delete('1.0', END)`

4. Додаємо до імпорту новий тип стандартного діалогу (`showinfo`):
`from tkinter.messagebox import showwarning, showinfo`
У тіло функції `calculate()` додаємо перевірку:

```
def calculate():
    try:
        r = float(radiusEntry.get())
    except:
        showwarning('Некоректні дані',
                    'Радіус має бути числом')
        return None
    rb = roundTo.get()
    text = ''
    l = 2*pi*r
    s = pi*r**2
    if rb != 'без заокруглення':
        l = round(l, int(rb))
        s = round(s, int(rb))
    if not calcL.get() and not calcS.get():
        showinfo('Некоректні параметри',
                 'Не вибрано, що обчислити')
    if calcL.get():
        text += f'Довжина кола:\n{l}\n\n'
    if calcS.get():
        text += f'Площа круга:\n{s}'
    result.delete('1.0', END)
    result.insert('1.0', text)
```



ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bader D. Python Tricks: The Book. – Dan Bader, 2017. – 302 p.
2. Hill C. Learning Scientific Programming with Python. – Cambridge University Press, 2015. – 450 p.
3. Lambert K. A. Fundamentals of Python: First Programs, 2nd Edition. – Cengage, 2019. – 498 p.
4. Lutz M. Learning Python, 5th Edition. – O'Reilly Media, 2013. – 1648 p.
5. Lutz M. Programming Python, 4th Edition. – O'Reilly Media, 2010. – 1632 p.
6. PEP 8 – Style Guide for Python Code. – https://www.python.org/dev/peps/pep-0008/
7. Python 3.x documentation – https://docs.python.org/3/
8. Sweigart A. Automate the Boring Stuff with Python: Practical Programming for Total Beginners. – No Starch Press, 2014. – 479 p.
9. Wentworth P., Elkner J., Downey A., Meyers C. How to Think Like a Computer Scientist: Learning with Python 3. – Green Tea Press, 2018. – 360 p.
10. Бейдер Д. Чистый Python. Тонкости программирования для профи / Д.Бейдер. – Санкт-Петербург : Питер, 2018. – 288 с.
11. Бэрри П. Изучаем программирование на Python / П. Бэрри. – Москва : Издательство "Э", 2017. – 624 с.
12. Федоров Д. Ю. Программирование на языке высокого уровня Python: учебное пособие для прикладного бакалавриата / Д. Ю. Федоров– Москва : Издательство Юрайт, 2017. – 126 с.

АЛФАВІТНИЙ ПОКАЖЧИК

А

Атрибут класу, 123

В

Виняток, 117

З

Змінна, 17
Зріз, 57, 75, 84, 168

І

Індекс, 57, 75, 83, 168
Інкапсуляція, 128
Інструкція, 15
  break, 49, 118, 172
  continue, 50
  else, 50
  if, 42
  return, 97
  try, 118
  with, 69
Інтерактивний режим, 7
Інтерпретатор, 6

К

Клас, 121
Ключові слова, 17
Коментар, 6
Композиція, 131
Конструктор класу, 125

Л

Літерал, 13
Лямбда-функція, 109

М

Модуль, 31
  fractions, 37
  math, 35
  matplotlib, 141
  random, 37
  time, 38
  tkinter, 144
  turtle, 137
  інші модулі, 39

О

Об’єктно-орієнтоване програмування, 121
Оператор, 14, 82

П

Перевантаження операторів, 134
Помилка
  виконання, 11
  семантична, 11
  синтаксична, 10
Псевдокод, 4

Р

Рядок символів, 55

С

Середовище розробки, 6
Символ
  керуючий, 56
Структура даних
  кортеж, 82
  множина, 89
  словник, 85
  список, 73
Сценарний режим, 8

Т

Точкова нотація, 31

У

Успадкування, 129

Ф

Файл, 65, 67, 71, 163
Функція, 20, 95
  input(), 23
  print(), 22
  range(), 46
  анотована, 109
  декоратор, 112
  рекурсивна, 107

Ц

Цикл
  for, 46
  while, 48

Ч

Число
  дійсне, 25
  комплексне, 26
  ціле, 25

ЗМІСТ

ПЕРЕДМОВА 3
РОЗДІЛ 1. МОВА ПРОГРАМУВАННЯ PYTHON ТА ЇЇ ОСОБЛИВОСТІ . 5
1.1. Загальні відомості про Python-програми 5
1.1.1. Структура програми 6
1.1.2. Коментування коду 6
1.1.3. Інтерактивний режим 7
1.1.4. Сценарний режим 8
1.2. Налагодження програми 10
1.2.1. Синтаксичні помилки..... 10
1.2.2. Помилки виконання..... 11
1.2.3. Семантичні помилки..... 11
Питання та завдання для самоперевірки..... 12
РОЗДІЛ 2. БАЗОВІ ПОНЯТТЯ ТА ТЕРМІНОЛОГІЯ 13
2.1. Основні терміни та поняття 13
2.1.1. Типи даних..... 13
2.1.2. Літерали..... 13
2.1.3. Вирази й оператори 14
2.1.4. Інструкції 15
2.1.5. Змінні..... 17
2.1.6. Функції..... 20
2.1.7. Об'єкти та класи 21
2.2. Засоби вводу-виводу інформації у Phyton..... 22
2.2.1. Функція виводу print()..... 22
2.2.2. Введення даних за допомогою функції input() .. 23
Питання та завдання для самоперевірки..... 24
РОЗДІЛ 3. ЧИСЛОВІ ТИПИ ДАНИХ МОВИ PYTHON 25
3.1. Цілі числа 25
3.2. Дійсні числа 25
3.3. Комплексні числа 26
3.4. Логічний тип 26

3.5. Операції з числовими типами даних 26
3.6. Вбудовані функції та методи для роботи з даними 27
Питання та завдання для самоперевірки 29
РОЗДІЛ 4. СТАНДАРТНА БІБЛІОТЕКА. МОДУЛІ ТА ЇХНЄ ВИКОРИСТАННЯ 31
4.1. Поняття модуля у Python 31
4.1.1. Точкова нотація 31
4.1.2. Способи імпортування модулів 33
4.2. Стандартна бібліотека Python 34
4.2.1. Модуль math 35
4.2.2. Модуль random 37
4.2.3. Модуль fractions 37
4.2.4. Модуль time 38
4.2.5. Інші модулі 39
Питання та завдання для самоперевірки 40
РОЗДІЛ 5. ОСНОВНІ КОНСТРУКЦІЇ МОВИ ТА ЇХНЯ РЕАЛІЗАЦІЯ .. 41
5.1. Логічні оператори та логічні вирази 41
5.2. Інструкція if 42
5.3. Безоператорні логічні вирази 45
5.4. Конструкції повторення (цикли) 45
5.4.1. Функція range() 46
5.4.2. Цикл for 46
5.4.3. Цикл while 48
5.4.4. Інструкція break 49
5.4.5. Інструкція continue 50
5.4.6. Блок else 50
5.5. Вкладені цикли 51
Питання та завдання для самоперевірки 52
РОЗДІЛ 6. РОБОТА З РЯДКАМИ ТА ФАЙЛАМИ 55
6.1. Рядки 55
6.1.1. Літерали рядків 55
6.1.2. Символи 55

6.1.3. Керуючі символи 56
6.1.4. Доступ до елементів рядка. Індекси та зрізи 57
6.1.5. Конкатенація (об'єднання) та повторення рядків 59
6.1.6. Форматування рядків..... 59
6.2. Функції для роботи з рядками та методи рядків 62
6.2.1. Перевірка на входження та посимвольний обхід рядка..... *65*
6.3. Файли..... 65
6.3.1. Запис у файл 66
6.3.2. Зчитування з файлу 67
6.3.3. Режими відкриття файлів 69
6.3.4. Інструкція with..... 69
6.3.5. Повне ім'я файлу 70
Питання та завдання для самоперевірки..... 71
РОЗДІЛ 7. ОСНОВНІ СТРУКТУРИ ДАНИХ PYTHON 73
7.1. Списки 73
7.1.1. Подання списків..... 73
7.1.2. Індекси та зрізи 75
7.1.3. Функції для роботи зі списками та методи списків 76
7.1.4. Перевірка на входження та поелементний обхід списку 77
7.1.5. Заповнення списків під час виконання програми 79
7.1.6. Генератори списків 79
7.1.7. Накладення імен *81*
7.2. Кортежі..... 82
7.2.1. Подання кортежів 82
7.2.2. Індекси та зрізи 83
7.2.3. Функції для роботи з кортежами та методи кортежів..... 84

7.2.4. Перевірка на входження та поелементний обхід кортежу 85
7.3. Словники 85
7.3.1. Подання словників *85*
7.3.2. Доступ до елементів словника *86*
7.3.3. Функції для роботи зі словниками та методи словників 87
7.3.4. Поелементний обхід словника 88
7.3.5. Заповнення словників під час виконання програми 88
7.3.6. Генератори словників 89
7.4. Множини 89
7.4.1. Подання множин 89
7.4.2. Операції над множинами 90
7.4.3. Функції для роботи з множинами та методи множин 91
7.4.4. Генерування множин 92
Питання та завдання для самоперевірки 93
РОЗДІЛ 8. ПРОЦЕДУРНЕ ПРОГРАМУВАННЯ 95
8.1. Функції та їх використання 95
8.1.1.Створення та виклик функції 95
8.1.2. Інструкція return 97
8.1.3. Повертання більше ніж одного значення 98
8.1.4. Області видимості, локальні та глобальні змінні 99
8.1.5. Позиційні та іменовані аргументи. Значення за замовчуванням 101
8.1.6. Передавання довільної кількості аргументів ... 103
8.1.7. Розпаковування аргументів 104
8.1.8. Декомпозиція 105
8.2. Рекурсивні функції 108
8.3. Анотування функцій 109

8.4. Лямбда-функції 110
8.5. Оперування функціями як об'єктами 111
8.6. Декоратори 113
Питання та завдання для самоперевірки 115
РОЗДІЛ 9. ЗАСОБИ ОПРАЦЮВАННЯ ВИНЯТКІВ 117
9.1. Основні типи винятків 117
9.1.1. Інструкція try 118
Питання та завдання для самоперевірки 120
РОЗДІЛ 10. ВСТУП В ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ НА PYTHON. ОСНОВНІ ЗАСАДИ ТА ПРИНЦИПИ 121
10.1. Поняття *клас* та *екземпляр класу* 121
10.2. Атрибути та методи класу 123
10.2.1. Конструктор класу 125
10.2.2. Метод __str__() 127
10.3. Інкапсуляція 128
10.4. Успадкування 129
10.5. Композиція 131
10.6. Перевантаження операторів 134
Питання та завдання для самоперевірки 136
РОЗДІЛ 11. ГРАФІЧНІ ІНСТРУМЕНТИ ТА ЗАСОБИ PYTHON 137
11.1. Графічний інструментарій 137
11.1.1. Модуль turtle 137
11.1.2. Візуалізація даних засобами matplotlib 141
11.2. Графічний інтерфейс користувача 143
11.2.1. Кореневе вікно 144
11.2.2. Написи 145
11.2.3. Поля для введення 146
11.2.4. Прапорці 147
11.2.5. Розміщення віджета у кількох рядках/стовпчиках 148
11.2.6. Контейнери 149
11.2.7. Перемикачі 149

11.2.8. Кнопки 151
11.2.9. Текстові поля 152
11.2.10. Опрацювання подій..... 153
11.2.11. Стандартні діалоги..... 155
Питання та завдання для самоперевірки 159
ВІДПОВІДІ ДО ПИТАНЬ І ЗАВДАНЬ ДЛЯ САМОПЕРЕВІРКИ 160
Відповіді до розділу 1 160
Відповіді до розділу 2 160
Відповіді до розділу 3 161
Відповіді до розділу 4 163
Відповіді до розділу 5 165
Відповіді до розділу 6 166
Відповіді до розділу 7 169
Відповіді до розділу 8 173
Відповіді до розділу 9 175
Відповіді до розділу 10 175
Відповіді до розділу 11 177
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 180
АЛФАВІТНИЙ ПОКАЖЧИК 181

Навчальне видання

Селіверстов Роман
Мельничин Андрій

ОСНОВИ ПРОГРАМУВАННЯ МОВОЮ PYTHON

Навчальний посібник

Редактор Наталка Плиса
Комп'ютерне макетування Андрій Мельничин
Обкладинка Світлана Корольчук

Формат 60x84/16.
Умовн. друк. арк
Наклад 200 прим. Зам. .

Львівський національний університет імені Івана Франка
79000, Львів, вул. Університетська, 1.

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців, виготовників і розповсюджувачів видавничої продукції.
Серія ДК № 3059 від 12.12.2007 р.