25. Елементами файлу є окремі слова. Записати в інший файл слова, які розпочинаються на літеру «о» або «а».

26. Елементами файлу є цілі числа. Видалити з нього число, записане після першого нуля (нулі у файлі обов'язково присутні). Результат записати в інший файл.

27. Два текстові файли однакового розміру, елементами яких є числа. Отримати третій файл, кожен елемент якого дорівнює: а) сумі відповідних елементів заданих файлів; б) більшому із відповідних елементів заданих файлів.

28. Два текстові файли однакового розміру, елементами яких є числа. Отримати третій файл, кожен елемент якого дорівнює: а) різниці відповідних елементів заданих файлів; б) меншому з відповідних елементів заданих файлів.

29. Два текстові файли однакового розміру, елементами яких є окремі літери. Отримати третій файл, кожен елемент якого є поєднанням відповідних літер першого і другого файлів.

30. Два текстові файли однакового розміру, елементами яких є окремі літери. Записати в третій файл всі співпадаючі елементи наявних файлів.

31. Елементами файлу є окремі символи (цифри та літери). Всі цифри цього файлу записати в другій файл, а решта символи – в третій файл. Порядок слідування зберігається.

Контрольні запитання

1. Визначити поняття «файл».

2. Назвіть види файлів, які використовують у Python.

3. Який файл називається текстовим? Яка максимальна довжина рядка в текстовому файлі? Чому в текстовому файлі використовують ознаку кінця рядка?

4. Назвіть інструкції створення, закриття та відкриття файлів. Чи можна текстовий файл відкрити одночасно для зчитування та запису?

Аудиторна робота

Приклади 13.1: файл

1. Відкриття файлу для зчитування

```
def read_file(fname):
 """Функція для зчитування файлу fname
 та виведення його вмісту на екран"""
 file=open(fname, 'r') # відкриття файлу для зчитування
 print('File '+fname+':') # виведення назви файлу
 # Зчитування вмісту файлу по рядкам
 for line in file:
  # Виведення рядка s
   print(line, end='')
file.close() # Закриття файлу
if __name__=='__main__':  read_file('data/file.txt')
```

Результат

```
File data/file.txt:
Lorem ipsum dolor sit amet, consectetur adipisicing elit. Cum,
```

```
dicta.
Nisi culpa beatae quaerat vitae consequatur rem distinctio fugiat,
blanditiis.
Nostrum nobis inventore ipsa distinctio doloremque maxime tempore
minus quos.
Quisquam repellat, ab corporis odio impedit officiis possimus
magni similique.
Modi, tenetur reiciendis dolor ut officiis repellendus totam
tempore deserunt.
```

2. Використання функції *os.path.join* для побудови шляху до файлу

```
# Модуль, який містить функції для роботи з шляхом у файловій
системі
import os.path

def read_file(fname):
      """Функція для зчитування файла fname
      та виведення його вмісту на екран"""
      file=open(fname,'r') # відкриття файлу для зчитування
      print('File '+fname+':') # виведення назви файлу
      # зчитування вмісту файла по рядках
      for line in file:
       print(line, end='')  # виведення рядка s
      file.close() # закриття файлу

if __name__=='__main__':
 # функція os.path.join з'єднує частини шляху у файловій системі
 # необхідним роздільником
   read_file(os.path.join('data', 'file.txt'))
```

3. Запис даних у текстовий файл

```
import os.path

text='''Hello!
I am a text file. And I had been written with a Python script
before you opened me, so look up the docs and try to delete
me using Python, too.'''

def write_text_to_file(filename, text):
 """Функція для запису у файл filename рядка text"""
 f=open(filename, "w")# відкриття файла для запису
 f.write(text) # Запис рядка text у файл
 f.close()# Закриття файлу

if __name__=='__main__':
 write_text_to_file(os.path.join('data','example02.txt'),text)
```

4. Використання оператора *with* для закриття файлу

```
import os.path

filename=os.path.join('data','file.txt') #побудова імені файлу
# Оператор with закриє файл по закінченню виконання
```

```
# операторів усередині нього або виникненні виключення
with open(filename) as file: print(file.read())
```

5. Відкриття текстового файла для зчитування з вказівкою кодування

__file__ – це атрибут модуля, в якому зберігається ім'я файлу його його вихідного коду

```
with open(__file__, 'r', encoding='utf-8-sig') as file:
  for number, line in enumerate(file):
   print('{0}\t{1}'.format(number+1,line),end='')
print()
```

6. Відкриття файлу для зчитування та запису

```
import os.path
import statistics
import datetime
def calculate_stats(filename):
  with open(filename, 'r+') as file:
    numbers=[float(line) for line in file.readlines()
      if line != '\n' and not line.lstrip().startswith('#')]

        sum_ = sum(numbers)
        mean = statistics.mean(numbers)
        median = statistics.median(numbers)
        cur_time = datetime.datetime.now()
        fmt = '\n' \
              '# Статистика від {time!s}\n' \
              '# Сума:    {sum}\n' \
              '# Медіана: {median}\n' \
              '# Середнє: {mean}'

        print(fmt.format(time=cur_time,
                         mean=mean,
                         median=median,
                         sum=sum_),
              file=file)
if __name__ == '__main__':
    filename = os.path.join('data', 'example05.txt')
    calculate_stats(filename)
```

7. Відкриття файла для дозапису

```
import os.path
import datetime

log_file=os.path.join('data', 'ex06_log.txt')
with open(log_file, 'a') as log:
    print(datetime.datetime.now(), file=log)
```

8. Перезапис файла

```
import os.path
filename=os.path.join('data', 'example07.txt')
# Зчитування файла
with open(filename, 'r') as file:
```

```
    lines=file.readlines()
# Модифікація даних
lines.insert(2, 'inserted line\n')
# Перезапис файла
with open(filename, 'w') as file:
    file.writelines(lines)
```

9. Використання файлового об’єкта *io.StringIO*

```
import io

# Створення потоку
stream=io.StringIO()  # или io.StringIO('початкове значення')
stream.write('asdf in memory') # Запис даних у потік

# отримання рядка із об’єкта StringIO
print(stream.getvalue())

# Виведення поточної позиції
print('Current position:', stream.tell())
stream.seek(0) # Перехід на початок потоку
stream.write('data') # Запис даних у потік

# Виведення поточної позиції
print('Current position:', stream.tell())
print(stream.read())# Зчитування даних, які є в потоці

# Виведення поточної позиції
print('Current position:', stream.tell())
# отримання рядка із об’єкта StringIO
print(stream.getvalue())
```

Результат

```
asdf in memory
Current position: 14
Current position: 4
 in memory
Current position: 14
data in memory
```

10. Використання бінарного файлу

```
from array import array
import os.path
prefix=os.path.join('data', 'ex09_')
numbers = list(range(300, 400)) # отримання списку чисел
# Запис у текстовий файл
with open(prefix+'text.txt','w') as txt_file:
   print(numbers, file=txt_file)
# Створення масива, який підтримує buffer_protocol, із списку
numbers_array=array('i', numbers)
# Запис у бінарний файл
binary_filename=prefix+'binary.bin'
with open(binary_filename, 'wb') as bin_file:
    bin_file.write(numbers_array)
```

```
# Підготовка масиву
filesize=os.path.getsize(binary_filename)  # розмір файла
int_len=array('i').itemsize # розмір одного елемента в байтах
read_array=array('i',(0 for _ in range(filesize // int_len)))

# Зчитування із бінарного файла
with open(binary_filename, 'rb') as file:
    file.readinto(read_array)  # зчитування у масив
print(read_array) # Виведення масиву на екран
# Перевірка, чи зчитані дані відповідають початковим
print(read_array.tolist() == numbers)
```

Результат

```
array('i', [300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310,
311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323,
324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336,
337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349,
350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362,
363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375,
376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388,
389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399])
True
```

11. Використання *json*. Серіалізація – десеріалізація.

Серіалізація – процес перетворення будь-якої структури даних у послідовність бітів; зворотною до операції серіалізації є операція *десеріалізації* – відновлення початкового стану структури даних з бітової послідовності.

Серіалізацію використовують для передачі об'єктів по мережі та для збереження їх у файли. Наприклад, потрібно створити розподілений додаток, різні частини якого мають обмінюватися даними зі складною структурою. У цьому випадку для типів даних, які передбачають передавати, пишеться код, який здійснює серіалізацію і десеріалізацію. Об'єкт заповнюється потрібними даними, потім викликають код серіалізації, в результаті виходить, наприклад, XML-документ. Результат серіалізації передається приймаючій стороні, наприклад, по електронній пошті або HTTP. Додаток-одержувач створює об'єкт того ж типу і викликає код десеріалізації, у результаті отримують об'єкт з тими ж даними, що були в об'єкті програми-відправника. За такою схемою працює, наприклад, серіалізація об'єктів через SOAP в Microsoft.NET.

```
import json
import os.path

data = [{
        'name': 'John',
        'age': 20,
    },
    {
        'name': 'Mary',
        'age': 19
    }]
filename = os.path.join('data', 'example10.json')
# Серіалізація
```

```
with open(filename, 'w') as file:
    json.dump(data, file)
# Десеріалізація
with open(filename, 'r') as file:
    read_data = json.load(file)
print(read_data)
```

Результат

```
[{'name': 'John', 'age': 20}, {'name': 'Mary', 'age': 19}]
```

12. Серіалізація за допомогою *pickle*. Декоратор

Декоратор reprlib.recursive_repr(fillvalue='...') – структурний шаблон проектування, призначений для динамічного підключення додаткових можливостей до об'єкта. Шаблон *Декоратор* надає гнучку альтернативу методу визначення підкласів з метою розширення функціональності.

```
import os.path
import pickle
import reprlib
class Person(object):
    """Клас, який описує людину"""
    def __init__(self, name, age, sibling=None):
        """Конструктор класу. Параметри:
            name    -- ім’я
            age     -- вік
            sibling -- брат або сестра  """
        self.name = name; self.age = age
        self.sibling = sibling
# Декоратор відслідковує рекурсивні виклики методу __repr__
# і не дає йому увійти в нескінченну рекурсію,
# повертаючи fillvalue замість викликів даного методу, які ще не завершені
    @reprlib.recursive_repr()
    def __repr__(self):
      """Подання об’єкта у рядках"""
      return 'Person({name!r}, {age!r}, {sibling!r})'.format(**self.__dict__)

def write_data(filename):
    """Функція створення і запису даних"""
    james=Person('James', 20); julia=Person('Julia', 21)
    james.sibling=julia  # створення циклічних посилань
    julia.sibling=james

    # Серіалізація списку об'єктів
    with open(filename, 'wb') as file:
    # 'wb' - запис бінарного файлу
        pickle.dump([james, julia], file)

def read_data(filename):
    """Функція зчитування і виведення даних на екран"""
    with open(filename, 'rb') as file:
        data = pickle.load(file)
```

```
        print(data)    # Виведення у консоль

    if __name__ == '__main__':
        filename = os.path.join('data', 'example11.pkl')
        write_data(filename); read_data(filename)
```

Результат

```
[Person('James',    20,    Person('Julia',    21,    ...)),
Person('Julia', 21, Person('James', 20, ...))]
```

Приклад 13.2: текстовий файл

Відкриття нового текстового файлу в режимі запису: в нього записують два рядки (що завершуються символом кінця рядка \n), після чого файл закривають. Далі цей файл відкривають в режимі зчитування і виконують зчитування рядків з нього. Третій виклик методу *readline* повертає порожній рядок – у такий спосіб методи файлів в Python повідомляють, що було досягнуто кінець файлу (порожній рядок у файлі повертають як рядок, що містить один символ нового рядка \n, а не як дійсно порожній рядок). Нижче наведено повний лістинг сеансу:

```
>>> myfile = open('myfile.txt', 'w')
# відкриття файлу (створення/очищення)
>>> myfile.write('hello text file\n') #Запис рядку тексту
16
>>> myfile.write('goodbye text file\n')
18
>>> myfile.close() # Виштовхує вихідні буфери на диск
>>> myfile = open('myfile.txt')
# відкриває файл: 'r' - за замовчуванням
>>> myfile.readline() # зчитування рядку
'hello text file\n'
>>> myfile.readline()
'goodbye text file\n'
>>> myfile.readline() # Пустий рядок: кінець файлу
```

Метод *write* повертає кількість записаних символів. Цей приклад записує два рядки тексту у файл, додаючи до кожного з них символ кінця рядка \n; методи запису не додають символ кінця рядка, тому необхідно самостійно додавати його у рядки, що виводяться (в іншому випадку наступна операція запису продовжить поточний рядок у файлі). Якщо необхідно вивести вміст файлу, забезпечивши правильну інтерпретацію символів кінця рядка, файл слід прочитати в рядок цілком, за допомогою методу *read*, і вивести:

```
>>> open('myfile.txt').read() #Прочитати файл в рядок цілком
'hello text file\ngoodbye text file\n'
>>> print(open('myfile.txt').read())
hello text file
goodbye text file
```

Якщо необхідно переглянути вміст файлу рядок за рядком, кращим вибором буде ітератор файлу:

```
    >>> for line in open('myfile'):
```

```
... print(line, end='')
hello text file
goodbye text file
```

Тут функція *open* створює тимчасовий об'єкт файлу, вміст якого автоматично буде зчитуватися ітератором і повертатися по одному рядку під час кожної ітерації циклу.

Приклад 13.3: текстовий файл

1. Записати різні об'єкти в текстовий файл

Дані завжди записують у файл у вигляді рядків, а методи запису не виконують форматування рядків:

```
>>> X,Y,Z = 43, 44, 45 # Об'єкти Python повинні записуватися
>>> S = 'Spam' # у файл тільки в вигляді рядків
>>> D = {'a': 1, 'b': 2}
>>> L = [1, 2, 3]
>>> F = open('datafile.txt', 'w') # Створює файл для запису
>>> F.write(S + '\n') # Рядки завершують символом \n
>>> F.write('%s,%s,%s\n' % (X, Y, Z)) # Перетворює числа в рядки
>>> F.write(str(L) + '$' + str(D) + '\n')
# Перетворює і розділяє символом $
>>> F.close()
```

2. Створивши файл, можна досліджувати його вміст (відкривши файл і прочитавши дані в рядок). Функція виведення дає побайтове подання вмісту, а інструкція *print* інтерпретує вбудовані символи кінця рядка, щоб забезпечити легке для зчитування відображення:

```
>>> chars = open('datafile.txt').read() # відображення рядка
>>> chars # у неформатованому вигляді
"Spam\n43,44,45\n[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> print(chars) # подання, яке зручно читати
Spam
43,44,45
[1, 2, 3]${'a': 1, 'b': 2}
```

Тепер необхідно виконати зворотні перетворення, щоб отримати з рядків у текстовому файлі об'єкти мови. Інтерпретатор Python не виконує перетворення рядків у числа або в об'єкти інших типів, тому необхідно виконати відповідні перетворення, щоб можна було використати операції над цими об'єктами, такі як індексування, додавання тощо:

```
>>> F = open('datafile.txt') # відкрити файл знову
>>> line = F.readline() # Прочитати один рядок
>>> line
'Spam\n'
>>> line.rstrip() # видалити  символ кінця рядка
'Spam'
```

Тут застосовано метод *rstrip*, щоб видалити символ кінця рядка. Цей результат можна отримати за допомогою вилучення зрізу line [:– 1], але такий підхід можна використовувати, якщо ви впевнені, що всі рядки завершуються символом \n (останній рядок в файлі іноді може не містити цей символ). Ми

прочитали частину файлу, який містить рядок. Тепер прочитаємо наступний блок, в якому містяться числа, і виконаємо розбиття цього блоку (тобто виокремимо об'єкти):

```
>>> line = F.readline() # наступний рядок з файлу
>>> line # це - рядок
'43,44,45\n'
>>> parts=line.split(',') # розбити на підрядки за комами
>>> parts
['43', '44', '45\n']
```

Тут використано метод *split*, щоб розбити рядок на частини за комами (кома – символ-роздільник). В результаті отримано список рядків, кожний з яких містить окреме число. Ці рядки необхідно перетворити в цілі числа, щоб виконувати математичні операції над ними:

```
>>> int(parts[1]) # Перетворити рядок у ціле число
44
>>> numbers = [int(P) for P in parts] # Перетворити весь список
>>> numbers
[43, 44, 45]
```

Функція *int* перетворює рядок цифр в об'єкт цілого числа, а генератор списків дозволяє застосувати функцію до всіх елементів списку в одній інструкції. Для видалення символа завершення \n в кінці останнього підрядка не застосовано метод *rstrip* (*int* та інші функції перетворення ігнорують символи-роздільники, які оточують цифри). Щоб перетворити список і словник в третьому рядку файлу, можна скористатися вбудованою функцією *eval*, яка інтерпретує рядок як програмний код на Python:

```
>>> line = F.readline()
>>> line
"[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> parts = line.split('$') #Розбити на рядки за символом $
>>> parts
['[1, 2, 3]', "{'a': 1, 'b': 2}\n"]
>>> eval(parts[0]) # Перетворити рядок у об'єкт
[1, 2, 3]
>>> objects=[eval(P) for P in parts]
# Теж саме для всіх рядків у списку
>>> objects
[[1, 2, 3], {'a': 1, 'b': 2}]
```

Оскільки тепер всі дані зображують собою список звичайних об'єктів, можна застосувати до них операції списків і словників.

Приклад 13.4. Сканування файлів.

Файли містять множину символів і рядків, тому їх можуть розглядати як один з типових об'єктів використання циклів. Щоб завантажити вміст файлу в рядок однією інструкцією, досить викликати метод *read*:

```
file = open('test.txt','r') #Прочитати вміст файлу в рядок
print(file.read())
```

Для завантаження файлу по частинах використовують або цикл *while*, який завершують інструкцією *break* після досягнення кінця файлу, або цикл *for*:

```
# цикл for обробляє окремо кожний символ, але завантаження вмісту
файлу в пам'ять виконують одноразово
     file = open('test.txt')
     while True:
          char = file.read(1) # зчитувати по одному символу
          if not char: break
          print(char)
     for char in open('test.txt').read():
          print(char)

# цикл while реалізує зчитування рядками або блоками
file = open('test.txt')
while True:
     line = file.readline() # зчитувати рядок за рядком
     if not line: break
     print(line, end=' ')
     # Прочитаний рядок вже містить символ \n
file=open('test.txt', 'rb')
while True:
     chunk = file.read(10) # зчитувати блоками по 10 байтів
     if not chunk: break
     print(chunk)
```

Двійкові дані зчитують блоками певного розміру. Однак в разі текстових даних зитування рядками за допомогою циклу *for* працює швидше:

```
for line in open('test.txt').readlines():
     print(line, end='')
for line in open('test.txt'):
# використання ітератора: найкращий спосіб  зчитування тексту
     print(line, end='')
```

Метод файлів *readlines* завантажує в список рядків цілий файл, тоді як при використанні ітератора файлу в кожній ітерації завантажують один рядок. Останній приклад – це найкращий спосіб роботи з текстовими файлами, він простіше і здатний працювати з файлами будь-якого розміру, тому що не завантажує цілий файл у пам'ять. Текстові файли за замовчуванням в процесі запису і зчитування відображають символи кінця рядка «\n», і виконують перетворення символів Юнікоду відповідно до кодування.

Приклад 13.5. Коли виконують операцію зчитування двійкових даних з файлу, вона повертає об'єкт типу *bytes* – послідовність коротких цілих чисел, що подають абсолютні значення байтів. Цей об'єкт нагадує звичайний рядок:

```
     >>> data=open('data.bin', 'rb').read()
     # відкриття двійкового файлу для зчитування
     >>> data # рядок bytes зберігає двійкові дані
     b'\x00\x00\x00\x07spam\x00\x08'
     >>> data[4:8] # веде себе як рядок
     b'spam'
     >>> data[4:8][0]
```

```
# Але в дійсності зберігає 8-бітові цілі числа
115
>>> bin(data[4:8][0]) # Функція bin()
'0b1110011'
```

Приклад 13.6. Скласти програму, яка підраховує кількість рядків і символів у файлі. В текстовому редакторі створіть модуль з ім'ям *mymod.py*, який експортує три імені: функцію

а) *countLines (name)*, яка зчитує вхідний файл і підраховує кількість рядків в ньому (підказка: більшу частину роботи можна виконати за допомогою методу *file.readlines*, а решту – за допомогою функції *len*).

б) *countChars (name)*, яка зчитує вхідний файл і підраховує кількість символів в ньому (підказка: метод *file.read* повертає один рядок).

в) *test(name)*, яка викликає дві попередні функції з заданим ім'ям файлу.

Ім'я файлу передають як аргумент функції. Всі три функції в модулі *mymod* повинні приймати ім'я файлу у вигляді рядка. Якщо розмір будь-якої з функцій перевищить два-три рядки, це означає, що ви робите зайву роботу.

Перевірте свій модуль в інтерактивній оболонці, використовуючи інструкцію *import* і повні імена експортованих функцій. Чи слід додати в змінну PYTHONPATH каталог, де знаходиться файл *mymod.py*? Спробуйте перевірити модуль, наприклад, таким чином *test*(*"mymod.py"*) – функція *test* відкриває файл двічі (спробуйте оптимізувати код, передаючи двом функціям підрахунку об'єкт відкритого файлу (підказка: метод *file.seek*(0) виконує переустановку вказівника на початок файлу).

Розв'язання. Необхідно сформувати файл *mymod.py* і сеанс взаємодії з інтерактивною оболонкою, як показано нижче (інтерпретатор Python може зчитувати вміст файлу цілком в список рядків, а отримати довжину кожного рядка в списку можна за допомогою вбудованої функції *len*):

mymod.py

```
def countLines(name):
 file=open(name)
 return len(file.readlines())
def countChars(name):
 return len(open(name).read())
def test(name): # або передати об'єкт файла
 return countLines(name),countChars(name) #або повернути словник
```

#main

```
>>> import mymod
>>> mymod.test('mymod.py')
(10, 291)
```

Ці функції зчитують цілий файл в пам'ять, а тому не в змозі працювати з великими файлами, які не можна вмістити в пам'яті комп'ютера (можна організувати зчитування вмісту файлу по рядках за допомогою ітератора і додавати довжину рядків кожного разу):

```
def countLines(name):
 tot = 0
```

```
 for line in open(name): tot += 1
 return tot
def countChars(name):
 tot = 0
 for line in open(name): tot += len(line)
 return tot
```

У Windows можна клацнути на файлі правою кнопкою миші і подивитися його властивості. Результат, який повертає сценарій, може відрізнятися від того, що повідомляє Windows, – інтерпретатор Python перетворює пару символів \r \n, які позначають кінець кожного рядка, в один символ \n, в результаті чого втрачають по одному байту (символу) на кожен рядок. Щоб результат сценарію в точності відповідав тому, що повідомляє Windows, файл необхідно відкрити в режимі двійкового доступу ('*rb*') або додавати до загального результату кількість рядків.

Щоб передати функціям об'єкт файлу для його відкриття лише один раз, використовують метод *seek* об'єкта файлу: він переустановлює поточну позицію в файлі у вказаний зсув. Після виклику методу *seek* наступні операції введення/виведення будуть виконуватися щодо нової позиції. Щоб переміститься у початок файлу, не закриваючи і не відкриваючи його повторно, можна викликати метод *file.seek*(0). Всі виклики методу *read* виконують зчитування з поточної позиції у файлі, тому, щоб розпочати повторне зчитування, необхідно перемістити поточну позицію на початок файлу. Нижче показано, як виглядає така реалізація:

```
def countLines(file):
 file.seek(0) # Перемістится на початок файлу
 return len(file.readlines())
def countChars(file):
 file.seek(0) # Те ж саме(перемістится на початок)
 return len(file.read())
def test(name):
 file = open(name) # Передати об’єкт файла
 return countLines(file), countChars(file)
# відкрити файл один раз
```

#main

```
>>> import mymod2
>>> mymod2.test("mymod2.py")
(11, 392)
```

Теоретичні відомості

1. ФАЙЛ

Файл – це іменована область пам'яті в комп'ютері, якою управляє операційна система (довільна послідовність елементів одного типу, довжина цих послідовностей заздалегідь не визначається, а конкретізується в процесі виконання програми; дані, що містяться у файлі, переносять на зовнішні носії).

Текстові файли прізначені для зберігання текстової інформації. Компоненти текстових файлів можуть мати змінну довжину.

Вбудована функція *open* створює об'єкт файлу, який забезпечує зв'язок з файлом, розміщеним в комп'ютері. Після виклику цієї функції можна виконувати операції зчитування і запису в зовнішній файл, використовуючи методи отриманого об'єкта. Об'єкти файлів не є ні числами, ні послідовностями або відображеннями – для роботи з файлами вони надають тільки методи. Більшість методів файлів пов'язані з виконанням операцій введення–виведення у зовнішні файли, асоційовані з об'єктом. У табл. 13.1 наведено операції над файлами, які найчастіше використовують.

Таблиця 13.1

Операції над файлами, які найчастіше використовують

Операція	Інтерпретація
output = open(r'C:\spam', 'w')	Відкриває файл для запису ('w' означає write – запис)
input = open('data', 'r')	Відкриває файл для зчитування ('r' означає read – зчитування)
input = open('data')	Відкриває файл для зчитування (режим 'r' використовують за замовчуванням)
aString = input.read()	Зчитування файлу цілком в один рядок
aString = input.read(N)	Зчитування наступних N символів (або байтів) в рядок
aString = input.readline()	Зчитування наступного текстового рядка (включаючи символ кінця рядка) в рядок
aList = input.readlines()	Зчитування файлу цілком в список рядків (включаючи символ кінця рядка)
output.write(aString)	Запис рядка символів (або байтів) у файл
output.writelines(aList)	Запис всіх рядків зі списку в файл
output.close()	Закриття файлу вручну (виконується після закінчення роботи з файлом)
output.flush()	Виштовхує вихідні буфери на диск, файл залишається відкритим
anyFile.seek(N)	Змінює поточну позицію в файлі для наступної операції, зсовуючи її на N байтів від початку файлу.
for line in open('data'): *операции над line*	Ітерації по файлу, зчитування по рядках
open('f.txt', encoding='latin-1')	Файли з текстом Юнікоду (рядки типу str)
open('f.bin', 'rb')	Файли з двійковими даними (рядки типу bytes)

Відкриття файлів. Щоб відкрити файл, програма повинна викликати функцію *open*, передавши їй ім'я зовнішнього файлу і режим роботи: як режим використовують рядок '*r*', якщо файл відкривають для зчитування (за замовчуванням), '*w*' – якщо файл відкривають для запису або '*a*' – для запису в кінець. У рядку режиму можна також зазначати інші параметри: додавання символу в рядок режиму означає 1) «*b*» – роботу з двійковими даними (відключають інтерпретацію символів кінця рядка і кодування символів Юнікоду); 2) «+» – файл відкривають для зчитування і для запису (є можливість зчитувати і записувати дані в один і той же об'єкт файлу, часто спільно з операцією позиціонування в файлі).

Обидва аргументи функції *open* повинні бути рядками. Крім того, функція може приймати третій необов'язковий аргумент, керуючий процесом буферизації виведених даних, – значення нуль означає, що вихідна інформація не буде буферизована (вона буде записуватися у зовнішній файл відразу ж, в момент виклику методу запису). Ім'я зовнішнього файлу може включати шлях до файлу, якщо шлях до файлу не вказано, передбачають, що файл розміщено в поточному робочому каталозі (тобто в каталозі, де був запущений сценарій).

Використання файлів. Як тільки отримано об'єкт файлу, можна викликати його методи для виконання операцій зчитування або запису.

Наведемо кілька основних зауважень щодо використання файлів:

1. *Для зчитування рядків краще використовувати ітератори файлів*.

Найкращий, мабуть, спосіб зчитування рядків з файлу на сьогоднішній день полягає в тому, щоб взагалі не використовувати операцію зчитування з файлу: файли мають ітератор, який автоматично зчитує інформацію з файлу рядок за рядком в контексті циклу *for*, в генераторах списків і в інших ітераційних контекстах.

2. *Вміст файлів знаходиться в рядках, а не в об'єктах.* Зверніть увагу: в табл. 13.1 показано, що дані, отримані з файлу, завжди потрапляють в сценарій у вигляді рядка. Якщо ця форма подання не підходить, необхідно виконати перетворення даних в інші типи об'єктів мови Python, а при виконанні операції запису даних у файл необхідно передавати методам сформовані рядки.

Тому при роботі з файлами треба згадати інструменти перетворення даних з рядка у число і навпаки (наприклад, *int, float, str*, а також вирази форматування рядків і метод *format*). Крім того, до складу Python входять додаткові стандартні бібліотечні інструменти, призначені для роботи з універсальним об'єктом «сховище даних» (наприклад, модуль *pickle*) і обробки упакованих двійкових даних у файлах (наприклад, модуль *struct*).

3. Виклик методу *close* є необов'язковим, він розриває зв'язок із зовнішнім файлом. Інтерпретатор Python негайно звільняє пам'ять, зайняту об'єктом, як тільки в програмі буде загублена останнє посилання на цей об'єкт. Як тільки об'єкт файлу звільняють, інтерпретатор закриває асоційований з ним файл (що відбувається також в момент завершення програми). Завдяки цьому не потрібно закривати файл вручну. З іншого боку, виклик методу *close* не зашкодить, і його рекомендують використовувати у великих системах (в момент закриття файлів звільняються ресурси операційної системи і виштовхуються вихідні буфери).

4. *Файли забезпечують буферизацію введення-виведення і дозволяють виробляти позиціонування у файлі.* За замовчуванням виведення у файли завжди виконують за допомогою проміжних буферів, тобто в момент запису тексту у файл він не потрапляє відразу ж на диск – буфери виштовхуються на диск тільки в момент закриття файлу або при виклику методу *flush*. Можна відключити механізм буферизації за допомогою додаткових параметрів функції *open*, але це може призвести до зниження продуктивності операцій введення-виведення. Файли в Python підтримують і можливість позиціонування – метод

seek дозволяє сценаріями управляти позицією зчитування і запису.

2. ТЕКСТОВІ ТА БІНАРНІ (ДВІЙКОВІ) ФАЙЛИ

В Python тип файлу визначається другим аргументом функції *open* – символ «*b*» в рядку режиму означає *binary* (двійковий). В Python існує підтримка текстових і двійкових файлів, але між цими двома типами файлів проведена чітка межа:

1. Вміст текстових файлів подають у вигляді звичайних рядків типу *str*, при цьому виконується автоматичне кодування/декодування символів Юнікоду, за замовчуванням проводиться інтерпретація символів кінця рядка.

2. Вміст бінарних файлів подають у вигляді рядків типу *bytes*, він передається програмі без будь-яких змін.

Звичайний текст і текст в Юнікоді об'єднані в один тип «рядок», адже будь-який текст можна подати в Юнікоді, включаючи ASCII та інші 8-бітові кодування. Більшості програмістам доводиться мати справу тільки з текстом ASCII, тому вони можуть користуватися базовим інтерфейсом доступу до текстових файлів і звичайними рядками. Всі рядки в Python 3.X є рядками Юнікоду, але для тих, хто використовує тільки символи ASCII, ця обставина зазвичай залишається непоміченою.

Для роботи з двійковими файлами слід використовувати рядки *bytes*, а звичайні рядки *str* – для роботи з текстовими файлами. Крім того, тому що текстові файли реалізують автоматичне перетворення символів Юнікоду, ви не зможете відкрити файл з двійковими даними в текстовому режимі – перетворення його вмісту в символи Юнікоду, швидше за все, завершиться з помилкою.

Збереження об'єктів Python за допомогою модуля pickle. Функція *eval* виконає будь-який вираз на мові Python. Якщо необхідно витягувати з файлів об'єкти Python, але не можна довіряти джерелу цих файлів, ідеальним рішенням є використати модуль *pickle*, який дозволяє зберігати у файлах будь-які об'єкти Python без необхідності виконувати їхнє перетворення. Щоб зберегти словник у файлі, наприклад, передамо його безпосередньо в функцію модуля *pickle*:

```
D = {'a': 1, 'b': 2}
F = open('datafile.pkl', 'wb')
import pickle
pickle.dump(D, F) # Модуль pickle запише у файл будь-який об'єкт
F.close()
```

Щоб потім прочитати словник назад, можна скористатися можливостями модуля *pickle*:

```
>>> F = open('datafile.pkl' 'rb')
>>> E = pickle.load(F) # Завантажує будь-які об'єкти з файлу
>>> E
{'a': 1, 'b': 2}
```

Отримали назад об'єкт–словник без необхідності виконати перетворення. Модуль *pickle* виконує перетворення об'єктів в рядок байтів і назад (тут

виконано перетворення словника в рядок):

```
>>> open('datafile.pkl', 'rb').read()
# Формат можна змінити
b'\x80\x03}q\x00(X\x01\x00\x00\x00aq\x01K\x01X\x01\x00\x00\x00bq\x
02K\x02u.'
```

У наведеному прикладі був відкритий файл, де зберігається об'єкт у двійковому режимі. В Python такі файли завжди слід відкривати саме в дійковому режимі, тому що модуль *pickle* створює і використовує об'єкти типу *bytes*. Модуль *shelve* – інструмент, який використовує модуль *pickle* для збереження об'єктів Python в файлах з доступом по ключу.

3. МОДУЛЬ

Для створення великих систем Python надає такі можливості, як модулі. Вони дозволяють розбити систему на складові. Термін «модуль» зарезервований для позначення файлів, які можуть імпортуватися іншими файлами. Кожен файл з вихідним текстом на мові Python, ім'я якого закінчується розширенням *.py*, є модулем. Інші файли можуть звертатися до програмних компонентів, які декларуються модулем, імпортуючи цей модуль. Інструкція *import* виконує завантаження іншого файлу і забезпечує доступ до його вмісту. Вміст модуля стає доступним зовнішнього світу через його атрибути. Така модульна модель є центральною ідеєю, яка лежить в основі архітектури програм на Python. Великі програми зазвичай організовані у вигляді множини файлів модулів, які імпортують і використовують функціональні можливості з інших модулів.

Один з модулів визначається як основний файл верхнього рівня, який запускає всю програму. Операція імпорту призводить до виконання програмного коду завантаження. Як наслідок, імпорт файлу є способом запустити його. Операція імпорту вимагає великих витрат обчислювальних ресурсів (в ході імпорту проводиться пошук файлів, компіляція їх в байт-код і виконання цього байт-коду).

Як організована програма. Як правило, програма на Python складається з множини текстових файлів, які містять інструкції. Програма організована як один головний файл, до якого можуть підключатися додаткові файли–модулі. Головний файл визначає, як буде рухатися основний потік виконання програми, – це той файл, який необхідно запустити, щоб розпочати роботу додатка. Файли модулів – це бібліотеки інструментальних засобів, де містяться компоненти, які використовує головний файл. Головний файл використовує інструменти, визначені у файлах модулів, а модулі використовують інструменти, визначені в інших модулях. В файлах модулів зазвичай визначені інструментальні засоби, які використовують в інших файлах. Щоб отримати доступ до визначених в модулі інструментів – атрибутів модуля (імен змінних, пов'язаних з такими об'єктами, як функції), необхідно імпортувати цей модуль: ми імпортуємо модулі і отримуємо доступ до їх атрибутів, що дозволяє використовувати їх функціональні можливості.

Комп'ютерний практикум № 14. ООП: створення класа та об'єктів–екземплярів класу

Мета роботи: ознайомитися з парадигмою об'єктно-орієнтованого програмування (ООП) на мові Python. *Об'єкт дослідження* – парадигма ООП, клас, екземпляр класу, конструктор класу.

ПЛАН

1. Парадигма об'єктно-орієнтованого програмування
2. Клас: створення класа, об'єктів-екземплярів класу
3. Конструктор класу – метод __init__.

Завдання

1. Ознайомитися з теоретичним матеріалом.
2. Відповідно до свого варіанту

- визначити умови; за допомогою формул описати варіанти виконання необхідний дій; розробити програмний додаток, який розв'язує завдання;
- організувати, якщо треба, введення даних з клавіатури і виведення у консоль;
- атрибути пов'язати з методами, які дозволяють їх змінювати
- реалізувати такі два *py* файли: *а*) створення класу, який не містить конструктор класу, в класі за допомогою атрибутів (поза функції) встановлюються декілька властивостей об'єктів; *б*) створення класу з конструктором классу (використати метод *__init__* для визначення початкових значень атрибутів об'єктів при їх створенні): для аргументів конструктора додати значення за замовчуванням;
- на основі класу створити декілька об'єктів-екземплярів.

3. Скласти звіт і захистити його по роботі.

Захист роботи включає в себе демонстрацію працездатності програми на різних вхідних даних.

Варіанти

1	Аудиторія у ВНЗ	2	Книга
3	Континент	4	Конференція
5	Автомобіль	6	Журнал
7	Овоч	8	Програмне забеспечення комп'ютера
9	Факультет інституту	10	Навчальна дисципліна
11	Газета	12	Геометрична фігура (трикутник, коло або прямокутник)
13	Місто	14	Фрукт
15	Міністерство охорони здоров'я	16	Торгівельний центр
17	Аптека	18	Операційна система (Linux, Windows, …)
19	Банк	20	Будівельні матеріали
21	Текстовий редактор	22	Лікарський препарат
23	Будинок	24	ЕОМ (комп'ютер)
25	Підприємство	26	Держава
27	Бібліотека	28	Група

Контрольні запитання

1. Назвіть парадигму об'єктно-орієнтованого програмування.
2. Яка інструкція передбачена для створення класів.
3. Для чого використовують метод конструктора *__init__*.

Аудиторна робота

Приклад 14.1. В Python все є об'єктами, в тому числі й самі класи

1. Створення екземпляра класу

```
# Оголошення порожнього класу  MyClass
class MyClass:  pass

obj = MyClass()
# Объект obj - це екземпляр класу MyClass,
# (він має тип MyClass)
print(type(obj))  # <class '__main__.MyClass'>
# MyClass – це клас, він є  об'єктом, екземпляром метакласу type
# який є абстракцією поняття типу даних
print(type(MyClass))  # <class 'type'>
# Тому з класами можна виконувати операції як із об'єктами
наприклад, копиювання
AnotherClass = MyClass
print(type(AnotherClass))
# тепер AnotherClass – це те ж саме, що і MyClass,
# і obj є екземпляром класу AnotherClass
print(isinstance(obj, AnotherClass))  # True
```

2. Всі елементи клада називають атрибутами.

Оголошення класу MyClass з двома атрибутами *int_field, str_field,* які є змінними,

```
class MyClass:
    int_field = 8
    str_field = 'a string'
# Звернення до атрибутів класу
print (MyClass.int_field); print (MyClass.str_field)
# Створення двох екземплярів  класу
object1 = MyClass (); object2 = MyClass ()
# Звернення до атрибутів класу через його екземпляри
print (object1.int_field); print (object2.str_field)
# Всі перераховані вище звернення до атрибутів насправді
відносяться
# до двох одних і тих самих  змінних
# Зміна значення атрибута класу
MyClass.int_field = 10
print (MyClass.int_field); print (object1.int_field); print
(object2.int_field)
# Однак, аналогічно до глобальних і локальних змінних,
# присвоєння значення атрибуту об'єкта не змінює значення
# атрибута класу, а веде до створення атрибута даних (нестатичного
поля)
object1.str_field = 'another string'
```

```
print(MyClass.str_field); print(object1.str_field);
print(object2.str_field)
```

3. Атрибути-дані аналогічні полям. Їх не треба описувати: як і змінні, вони створюються в момент першого присвоювання.

```
# Клас, який описує людину
class Person: pass
# Створення екземплярів  класу
alex = Person()
alex.name = 'Alex';  alex.age = 18

john = Person()
john.name = 'John'; john.age = 20

# Атрибути-дані відносять тільки до окремих екземплярів класу
# і ніяк не впливають на значення відповідних атрибутів-даних
інших екземплярів
print(alex.name, 'is', alex.age); print(john.name, 'is', john.age)
```

4. Атрибутами класу можуть бути й методи-функції

```
# Клас,  який описує  людину
class Person:
    # Перший  аргумент,  який  вказує  на поточний екземпляр
класу,
    # прийнято називати  self
    def print_info(self):
        print(self.name, 'is', self.age)

# Створення екземплярів класу
alex=Person(); alex.name='Alex'; alex.age=18

john = Person(); john.name = 'John'; john.age = 20

# Перевіримо, чим є  атрибут-функція print_info класу Person
print(type(Person.print_info))  # функція (<class 'function'>)

# Викличемо його  для об’єктів alex і john
Person.print_info(alex)
Person.print_info(john)

# Метод – функція, зв’язана з об’єктом. Всі атрибути класу, які є
# функціями, описують відповідні методи екземплярів даного класу
print(type(alex.print_info))  # метод (<class 'method'>)

# Виклик  методу print_info
alex.print_info()
john.print_info()
```

5. Початковий стан об’єкта слід створювати в методі-конструкторі __init__, який викликають автоматично після створення екземпляру класа. Його параметри вказують при створенні об’єкта.

```
# Клас, який описує людину
class Person:
    # Конструктор
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # Метод з попереднього прикладу
    def print_info(self):
        print(self.name, 'is', self.age)
# Створення екземплярів классу
alex = Person('Alex', 18)
john = Person('John', 20)
# Виклик метода print_info
alex.print_info()
john.print_info()
```

6. Атрибути класу, які є функціями, – це тіж самі атрибути класу, як і змінні

```
def outer_method(self):
    print('I am a method of object', self)
class MyClass:
    method = outer_method

obj = MyClass(); obj.method()
```

7. Статичними називають методи, які є загальними для класу та усіх екземплярів класу і не мають доступ до даних екземплярів класів. Для їх створення використовують декоратор *staticmethod.* Декоратор – це спеціальна функція, яка змінює поведінку функції або класу. Для використання декоратора треба перед відповідним оголошенням вказати символ @, ім'я необхідного декоратора і список його аргументів в круглих дужках. Якщо передавати параметрів декораторові не потрібно, дужки не вказують.

```
class MyClass:
    # Оголошення атрибута класу
    class_attribute = 8

    def __init__(self):  # Конструктор
        self.data_attribute = 42

    # Статичний метод (у нього немає параметру) self,
    # оскільки він не зв’язаний з жодним із екземплярів класу
    # не має доступу до  атрибутівм-даних
    @staticmethod
    def static_method():
        print(MyClass.class_attribute)

    def instance_method(self): # звичайний  метод
        print(self.data_attribute)

if __name__ == '__main__':
    # Виклик статичного методу
    MyClass.static_method()
```

```
    # Інстанціювання об’єкта
    obj = MyClass()
    # Виклик методу
    obj.instance_method()
    # Аналогічно атрибутам класу, доступ до статичних  методів
    # можна отримати й через екземпляр класу
    obj.static_method()
```

8. Класи є об'єктами, тому крім атрибутів-функцій вони можуть мати і власні методи. Для створення методів класу використовують декоратор *classmethod*. В таких методах перший параметр прийнято називати не *self*, а *cls*. Методи класу зазвичай використовують в двох випадках: для створення
• фабричних методів, які створюють екземпляри даного класу альтернативними способами;
• статичних методів, які викликають статичні методи:
- оскільки даний клас передається як перший аргумент функції,
- не потрібно вручну вказувати ім'я класу для виклику статичного методу.

```
class Rectangle:
    """    Клас, який описує прямокутник     """
    def __init__(self, side_a, side_b):
        """  Конструктор класу
        :param side_a: перша сторона
        :param side_b: друга сторона       """
        self.side_a = side_a
        self.side_b = side_b

    def __repr__(self):
     """Метод, який повертає подання об'єкта у вигляді рядка """
      return 'Rectangle(%.1f, %.1f)' % (self.side_a, self.side_b)

class Circle:
    """ Клас, який описує коло """
    def __init__(self, radius):
        self.radius = radius
    def __repr__(self):
        return 'Circle(%.1f)' % self.radius

    @classmethod
    def from_rectangle(cls, rectangle):
        """Ми використовуємо метод класу в якості фабричного
методу,
        який створює екземпляр класу Circle з екземпляру
        класу Rectangle як коло, що вписане у цей  прямокутник.
        :param rectangle: Rectangle instance
        :return: Circle instance       """
        radius = (rectangle.side_a ** 2 + rectangle.side_b ** 2) ** 0.5 / 2
        return cls(radius)

def main():
    rectangle = Rectangle(3, 4)
```

```
    print(rectangle)
    circle1 = Circle(1)
    print(circle1)
    circle2 = Circle.from_rectangle(rectangle)
    print(circle2)

if __name__ == '__main__':  main()
```

9. Атрибути, імена яких розпочинаються, але не закінчуються, двома символами підкреслення, вважаються приватними. До них застосовують механізм «name mangling»: зсередини класу і його екземплярів до цих атрибутів можна звертатися по тому ж імені, яке було задано при оголошенні (однак до імен зліва додається підкреслення і ім'я класу). Цей механізм не передбачає захисту даних від зміни ззовні, тому що до них все одно можна звернутися, знаючи ім'я класу і те, як Python змінює імена приватних атрибутів, проте дозволяє захистити їх від випадкового перевизначення в класах-нащадках.

```
class MyClass:
    def __init__(self):
        self.__private_attribute = 42

    def get_private(self):
        return self.__private_attribute

obj = MyClass()
print(obj.get_private())  # 42
print(obj.__private_attribute)  # помилка
# print(obj._MyClass__private_attribute)  # 42
```

10. Атрибути, імена яких розпочинаються і закінчуються двома знаками підкреслення, є внутрішніми для Python і задають особливі властивості об'єктів (наприклад, рядок документування *__doc__*, атрибут *__class__*, в якому зберігають клас даного об'єкта). Серед таких атрибутів є методи. Подібні методи називають методами зі спеціальними іменами: вони задають особливу поведінку об'єктів і дозволяють перевизначати поведінку вбудованих функцій і операторів для екземплярів заданого класу. Найчастіше використовують метод-конструктор *__init__*.

```
class Complex:
    """    Комплексне число    """
    def __init__(self, real=0.0, imaginary=0.0):
        """ Конструктор
        :param real:      дійсна частина
        :param imaginary: уявна частина  """
        self.real = real; self.imaginary = imaginary

    def __repr__(self):
        """  Метод  __repr__  повертає подання об’єкта у вигляді
рядка, який має вигляд виразу, що створює  аналогічний об’єкт,
інакше містить його  опис;
        Викликають функцією repr. """
        return 'Complex(%g, %g)' % (self.real, self.imaginary)
```

```
    def __str__(self):
        """ Метод __str__ повертає подання об'єкта у вигляді
рядка; його викликають функції str, print і format.  """
        return '%g %c %gi' % (self.real,
                              '+' if self.imaginary >= 0 else '-',
                              abs(self.imaginary))

    # Арифметичні операції
    def __add__(self, other):
        """  Метод __add__ визначає  операцію додавання. """
        return Complex(self.real + other.real,
                self.imaginary + other.imaginary)

    def __neg__(self):
        """  Операція заперечення """
        return Complex(-self.real, -self.imaginary)

    def __sub__(self, other):
        """  Операція віднімання.
        Додавання і заперечення вже визначені, тому віднімання
        можна визначити через них  """
        return self + (-other)

    def __abs__(self):
        """  Модуль числа  """
        return (self.real ** 2 + self.imaginary ** 2) ** 0.5

    # Операції порівняння
    def __eq__(self, other):
     return self.real==other.real and self.imaginary==other.imaginary

    def __ne__(self, other):
        return not (self == other)

def main():
    x = Complex(2, 3.5)
    print(repr(x)); print('x =', x)
    y = Complex(5, 7)
    print('y =', y); print('x + y =', x + y)
    print('x - y =', x - y); print('|x| =', abs(x))
    print('(x == y) =', x == y)

if __name__ == '__main__':  main()
```

11. Використання спеціального методу *__new__* для реалізації такого шаблону проектування як Одинак (Singleton) (це шаблон проектування, який гарантує, що даний клас має тільки один екземпляр, і породжує його).

```
class Singleton:
  _instance = None  # атрибут, який зберігає екземпляр класу

  def __new__(cls, *args, **kwargs):
   """ Метод __new__ викликають при створенні екземпляра класу """
```

```
   # Якщо екземпляр ще не створений, то створюємо його
   if cls._instance is None:
       cls._instance = object.__new__(cls, *args, **kwargs)
       # Повертаємо екземпляр, який існує
   return cls._instance

  def __init__(self):
   self.value = 8

obj1=Singleton(); print(obj1.value)

obj2 = Singleton(); obj2.value = 42
print(obj1.value)
```

12. Використання функції __getattribute__ для приховування даних

```
class MyClass:
   def __init__(self):
        self.password = None

   def __getattribute__(self, item):
    """Метод __getattribute__ викливають при отриманні
атрибутів"""

        # Якщо поле-запит secret_field і пароль правильний
        if item == 'secret_field' and self.password == '9ea)fc':
            # то повертаємо значення
            return 'secret value'
        else:
            # инакше викликаємо метод __getattribute__ класу
object
            return object.__getattribute__(self, item)

obj = MyClass()# Створення екземпляру класу
# Розблокування секретного поля
# obj.password = '9ea)fc'
# Виведення значення secret field.
# Значення буде отримано, якщо  розкоментувати попередній
# рядок  програмного коду, інакш отримаємо помилку
print(obj.secret_field)
```

Приклад 14.2. Створимо клас з одним атрибутом поза методу й одним методом, який виводить значення цього атрибута на екран із змінами:

```
class First:
 color = "red"
 def out(self):  print (self.color + "!")
```

Створимо два об’єкти даного класу:

```
obj1 = First(); obj2 = First()
# Обидва об’єкта (obj1, obj2) мають  два однакових атрибута:
# color -  у вигляді властивості й printer (у вигляді методу)
print (obj1.color); print (obj2.color)
obj1.out(); obj2.out()
```

В результаті отримано:

```
"red"
"red"
"red!"
"red!"
```

Нехай в класі за допомогою атрибутів визначені дві властивості об'єктів: червоний колір і кругла форма. Методи можуть змінювати ці властивості в залежності від побажань тих, хто створює об'єкти.

```
class Second:
 color = "red"
 form = "circle"
 def changecolor(self,newcolor):
  self.color = newcolor
 def changeform(self,newform):
  self.form = newform

obj1 = Second(); obj2 = Second()
print (obj1.color, obj1.form) # вывод на экран "red circle"
print (obj2.color, obj2.form) # вывод на экран "red circle"

obj1.changecolor("green") # изменение цвета первого объекта
obj2.changecolor("blue")  # изменение цвет второго объекта
obj2.changeform("oval")  # изменение формы второго объекта
print (obj1.color, obj1.form) # вывод на экран "green circle"
print (obj2.color, obj2.form) # вывод на экран "blue oval"
```

Тут за замовчуванням будь-який створений об'єкт має червоний колір і круглу форму. Однак в подальшому за допомогою методів даного класу можна поміняти і колір і форму будь-якого об'єкта. В результаті об'єкти перестають бути однаковими (червоними і круглими), хоча зберігають однаковий набір властивостей (колір і форму).

Як же відбуваються зміни? – Методи крім параметра *self*, можуть мати й інші параметри, в яких передаються дані для обробки їх цим методом. Наприклад, метод *changecolor* має додатковий параметр newcolor, за допомогою якого, в метод можна передати дані про бажані кольори фігури. Далі метод змінює колір за допомогою відповідних інструкцій.

Приклад 14.3. Більшість класів використовують спеціальний метод *__init__*, який при створенні об'єкта створює йому атрибути (викликати даний метод не потрібно, він сам запускається під час виклику класу, а виклик класу відбувається, коли створюється об'єкт). Такий метод називають конструктором класу. Першим параметром у *__init__* є *self*, на місце якого підставляється об'єкт в момент його створення. Другий і наступні (якщо є) параметри замінюють аргументами, які передані в конструктор при виклику класу. Розглянемо два класи: в одному використовується конструктор, а в іншому ні. Потрібно створити два атрибути об'єкта.

```
1)class YesInit:
 def __init__(self,one,two):
```

```
  self.fname = one; self.sname = two

obj1 = YesInit("Peter","Ok")
print (obj1.fname, obj1.sname)

2)class NoInit:
 def names(self,one,two):
  self.fname = one; self.sname = two

obj1 = YesInit(); obj1.names("Peter","Ok")
print (obj1.fname, obj1.sname)
```

Результат роботи програми в обох випадках:

```
Peter Ok
```

В обох програмах у об'єкта з'являються два атрибути: *fname* і *sname*. *У першій програмі* вони приймають початкові значення при створенні об'єкта і повинні передаватися в дужках при виклику класу. Якщо атрибути повинні бути присутніми у об'єктів класу обов'язково, то використання методу *__init__* – ідеальний варіант. *У другій програмі* (без використання конструктора) атрибути створюють шляхом виклику методу *names* після створення об'єкта. В даному випадку виклик методу *names* необов'язковий, тому об'єкти можуть існувати без атрибутів *fname* і *sname*.

Зазвичай метод *__init_* передбачає передачу аргументів при створенні об'єктів, однак аргумент може не бути переданий. Наприклад, якщо в прикладі вище створити об'єкт так: *obj*1 = *YesInit*(), тобто не передати класу аргументи, то станеться помилка. Щоб уникнути подібних ситуацій, можна в методі *__init_* привласнювати параметрам значення за замовчуванням. Якщо при виклику класу були задані аргументи для даних параметрів, то вони і будуть використані, якщо ні – в тілі методу будуть використані значення за замовчуванням. Наприклад:

```
class YesInit:
 def __init__(self,one="noname",two="nametoo"):
  self.fname = one; self.sname = two
obj1 = YesInit("Sasha","Tu");obj2 = YesInit()
obj3 = YesInit("Spartak"); obj4 = YesInit(two="Harry")
print (obj1.fname,obj1.sname); print (obj2.fname,obj2.sname)
print (obj3.fname,obj3.sname); print (obj4.fname, obj4.sname)
```

Результат роботи:

```
Sasha Tu
noname nametoo
Spartak nametoo
noname Harry
```

Тут другий об'єкт створюють без передачі аргументів, тому в методі *__init__* використовують значення за замовчуванням ("*noname*", "*nametoo*"). При створенні третього і четвертого об'єктів передають по одному аргументу.

Якщо вказують значення не першого аргументу, то слід явно вказати ім'я параметра (четвертий об'єкт). Метод *__init_*_ може містити параметри як без значень за замовчуванням, так і зі значеннями за замовчуванням. В цьому випадку параметри, аргументи яких повинні бути обов'язково вказані, при створенні об'єктів вказують першими, а параметри зі значеннями за замовчуванням – після, наприклад:

```
class fruits:
 def __init__(self,w,n=0):
  self.what = w; self.numbers = n

f1 = fruits("apple",150); f2 = fruits("pineapple")
print (f1.what,f1.numbers); print (f2.what,f2.numbers)
```

Створимо клас, значення початкових атрибутів (з методу *__init__*) якого залежить від переданих аргументів при створенні об'єктів. Далі ці атрибути об'єктів, створених на основі даного класу, можна змінювати за допомогою методів.

```
class Building:
 def __init__(self,w,c,n=0):
  self.what=w; self.color=c; self.numbers=n; self.mwhere(n)

 def mwhere(self,n):
  if n <= 0:  self.where = "отсутствуют"
  elif 0 < n < 100: self.where = "малый склад"
  else: self.where = "основной склад"

 def plus(self,p):
  self.numbers = self.numbers + p
  self.mwhere(self.numbers)

 def minus(self,m):
  self.numbers=self.numbers-m
  self.mwhere(self.numbers)

m1 = Building("доски", "белые",50)
m2 = Building("доски", "коричневые", 300)
m3 = Building("кирпичи","белые")
print (m1.what,m1.color,m1.where)
print (m2.what,m2.color,m2.where)
print (m3.what,m3.color,m3.where)
m1.plus(500);print (m1.numbers, m1.where)
```

Тут значення атрибута *where* об'єкта залежить від значення атрибута *numbers*.

Теоретична частина

1. Парадигма об'єктно-орієнтованого програмування. Багато сучасних мов підтримують кілька парадигм програмування (наприклад, директивне, функціональне, об'єктно-орієнтоване). Такі мови є змішаними. До них

відносять і Python. Ключову роль в ООП відіграє множина об'єктів. Реальний світ складається з об'єктів та їхньої взаємодії між собою. В результаті взаємодії об'єкти можуть змінюватися самі або змінювати інші об'єкти. У світі умовно можна виділити різні системи, які реалізують певні цілі (змінюються з одного стану в інший). Наприклад, група на занятті – це система, яка складається з таких об'єктів, як діти, учитель, столи, комп'ютери, проектор тощо. У цієї системи можна виділити основну мету – збільшення частки знань дітей на якусь величину. Щоб домогтися цього, об'єкти системи повинні певним чином виконати взаємодію між собою.

Необхідно розуміти різницю між програмою написаною на основі структурного «стилю» і програмою в «стилі» ООП. У першому випадку, на перший план виходить логіка, розуміння послідовності виконання виразів (дій) для досягнення цілей. У другому – важливо системне мислення, вміння бачити систему в цілому, з одного боку, і розуміння ролі її частин (об'єктів), з іншого.

Свого часу Алан Кей сформулював для розробленої ним мови програмування Smalltalk кілька принципів, які описують принципи ООП:

1) об'єктно-орієнтована програма складається з об'єктів, які посилають один одному повідомлення;

2) кожний об'єкт може складатися з інших об'єктів (а може і не складатися);

3) кожний об'єкт належить певному класу (типу), який задає поведінку об'єктів, створених на його основі.

2. Клас – це опис об'єктів певного типу. На основі класів створюють об'єкти. Може існувати множина об'єктів, які належать одному класу. З іншого боку, може існувати клас без об'єктів, реалізованих на його основі. Програма, написана з використанням парадигми ООП, повинна складатися з: об'єктів, класів (опису об'єктів), взаємодій об'єктів між собою, в результаті яких змінюються їх властивості. Об'єкт в програмі можна створити лише на основі будь-якого класу. Тому, ООП має розпочинатися з проектування і створення класів. Класи можуть бути розташовані або спочатку коду програми, або імпортуватися з інших файлів – модулів (також на початку коду).

Створення класів. Для створення класів передбачена інструкція *class*, яка складається з рядка заголовка і тіла. Заголовок складається з ключового слова *class*, імені класу і, можливо, назв суперкласів в дужках. Суперкласи можуть бути відсутніми, в такому випадку дужки не потрібні. Тіло класу складається з блоку різних інструкцій. Тіло повинно мати відступ (як і будь-які вкладені конструкції в Python). Схематично клас можна подати таким чином:

```
class ІМ'Я_КЛАСУ:
 змінна = значення  ….
 def ІМ'Я_МЕТОДА(self, ...):
  self.змінна = значення …
```

У заголовку після імені класу можна вказати суперкласи (в дужках), а методи можуть бути більш складними. *Методи в класах* – це звичайні функції, за одним винятком: вони мають один обов'язковий параметр – *self*, який

потрібен для зв'язку з конкретним об'єктом. *Атрибути класу* – це імена змінних поза функцій і імена функцій. Вони успадковуються всіма об'єктами, створеними на основі даного класу, і забезпечують властивості і поведінку об'єкта. Об'єкти можуть мати атрибути, які створюються в тілі методу, якщо даний метод буде викликано для конкретного об'єкта.

Створення об'єктів. Об'єкти створюються так:

змінна = ІМ'Я_КЛАСУ()

Після такої інструкції у програмі з'являється об'єкт, доступ до якого можна отримати за ім'ям змінної, пов'язаної з ним. При створенні об'єкт отримує атрибути його класу (він має характеристики, визначені у його класі). Кількість об'єктів, які можна створити на основі певного класу, не обмежена. Об'єкти одного класу мають схожий набір атрибутів, а значення атрибутів можуть бути різними (вони схожі, але індивідуально різняться).

Self: Методи класу − це невеликі програми, призначені для роботи з об'єктами. Методи можуть створювати нові властивості (дані) об'єкта, змінювати існуючі, виконувати інші дії над об'єктами. Методу необхідно «знати», дані якого об'єкта йому потрібно буде обробляти. Для цього йому в якості першого (а іноді і єдиного) аргументу передається ім'я змінної, пов'язаної з об'єктом. Щоб в описі класу вказати об'єкт, який передається в подальшому, використовують параметр *self*. Виклик методу для конкретного об'єкта в основному блоці програми виглядає таким чином: ОБ'ЄКТ.ІМ'Я_МЕТОДА(…), де ОБ'ЄКТ – змінна, пов'язана з ним. Цей вираз перетворюється в класі, до якого відноситься об'єкт, в ІМ'Я_МЕТОДА(ОБ'ЄКТ, …) – замість параметра *self* підставляють конкретний об'єкт. Атрибути екземпляра створюються шляхом присвоювання значень атрибутам об'єкта екземпляра. Зазвичай вони створюються всередині методів класу, в інструкції *class* – присвоєнням значень атрибутам аргументу *self* (який завжди є імовірним екземпляром). Можна створювати атрибути за допомогою операції присвоєння в будь-якому місці програми, де доступне посилання на екземпляр, навіть за межами інструкції *class*.

3. Конструктор. Зазвичай всі атрибути екземплярів ініціалізують в конструкторі *__init__*. Метод конструктора *__init__* використовують для встановлення початкових значень атрибутів екземплярів і виконання інших початкових операцій (це − звичайна функція, яка підтримує і можливість визначення значень аргументів за замовчуванням, і передачу іменованих аргументів).