МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ МІСЬКОГО ГОСПОДАРСТВА імені О. М. БЕКЕТОВА

І. С. Творошенко

СПЕЦІАЛІЗОВАНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

КОНСПЕКТ ЛЕКЦІЙ

(для магістрів денної та заочної форм навчання спеціальності 193 – Геодезія та землеустрій освітньої програми «Геодезія та землеустрій»)

Харків
ХНУМГ ім. О. М. Бекетова
2018

УДК 004.432.42

Творошенко І. С. Спеціалізоване програмне забезпечення : конспект лекцій для магістрів денної та заочної форм навчання спеціальності 193 – Геодезія та землеустрій освітньої програми «Геодезія та землеустрій» / І. С. Творошенко ; Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. – Харків : ХНУМГ ім. О. М. Бекетова, 2018. – 118 с.

Автор
канд. техн. наук, доц. І. С. Творошенко

Рецензенти:

І. В. Шостак, доктор технічних наук, професор, професор кафедри інженерії програмного забезпечення (Національний аерокосмічний університет імені М. Є. Жуковського «ХАІ»);

О. А. Винокурова, доктор технічних наук, професор, головний науковий співробітник проблемної науково-дослідної лабораторії (Харківський національний університет радіоелектроніки)

Рекомендовано кафедрою земельного адміністрування та геоінформаційних систем, протокол № 1 від 30.08.2017.

Конспект лекцій складено з метою надати теоретичний матеріал щодо особливостей спеціалізованого програмного забезпечення та приклади застосування його на практиці для магістрів спеціальності «Геодезія та землеустрій», що допоможуть під час підготовки до практичних занять, виконання розрахунково-графічного завдання, складання екзамену з дисципліни «Спеціалізоване програмне забезпечення».

ЗМІСТ

ВСТУП…….…….. 4
1 ОСНОВИ ПРОГРАМУВАННЯ НА МОВІ PYTHON……... 5
1.1 Введення в програмування мовою Python...……..…… 5
1.2 Типи даних у мові програмування Python…….. 9
2 ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ АЛГОРИТМІВ
НА МОВІ ПРОГРАМУВАННЯ PYTHON...……...……..... 36
2.1 Алгоритмічні структури у мові програмування Python...……. 36
2.2 Модулі та пакети у мові програмування Python…… 64
3 РОЗРОБКА ДОДАТКІВ З ВИКОРИСТАННЯМ
МОВИ ПРОГРАМУВАННЯ PYTHON……...……...…… 74
3.1 Об’єктно-орієнтований підхід у мові програмування Python...…… 74
3.2 Програмування додатків баз даних на мові програмування Python… 93
СПИСОК РЕКОМЕНДОВАНИХ ДЖЕРЕЛ……... 118

ВСТУП

Конспект лекцій з дисципліни «Спеціалізоване програмне забезпечення» складено з метою надати теоретичний матеріал та приклади застосування його на практиці від основ до поглиблених тем для магістрів спеціальності «Геодезія та землеустрій» усіх форм навчання.

Метою викладання навчальної дисципліни «Спеціалізоване програмне забезпечення» є навчити студентів самостійно будувати програми різної складності мовою програмування Python за допомогою використання структурно-модульного методу програмування.

Завданням дисципліни «Спеціалізоване програмне забезпечення» є вивчення принципів використання мови програмування Python, а також засвоєння практичних аспектів побудови базових алгоритмів та програм різного рівня складності.

У результаті вивчення навчальної дисципліни магістр повинен мати компетентності пов’язані із здатністю:

– збирати, обробляти та інтерпретувати дані сучасних наукових досліджень, що необхідні для формування висновків по відповідним науковим дослідженням;

– розуміти, удосконалювати та застосовувати сучасний математичний апарат, фундаментальні концепції та системні методології, міжнародні та професійні стандарти в області інформаційних технологій;

– застосовувати геоінформаційні технології для аналізу, моделювання та програмування просторових об’єктів і явищ, розробляти геоінформаційні системи різного призначення;

– продукувати нові ідеї, проявляти креативність та здатність до системного мислення;

– використовувати базові знання природничих наук, математики та інформатики, основні факти, концепції, принципи теорій, пов’язаних із фундаментальною інформатикою та інформаційними технологіями;

– ефективно застосовувати базові математичні знання та інформаційні технології під час вирішення проектно-технічних і прикладних задач, пов’язаних із розвитком і використанням інформаційних технологій;

– застосовувати знання спеціалізованого програмного забезпечення і геоінформаційних систем, базові вміння програмувати для вирішення прикладних професійних задач.

1 ОСНОВИ ПРОГРАМУВАННЯ НА МОВІ PYTHON

1.1 Введення в програмування мовою Python

План

1. Поняття «програма» та «мова програмування».
2. Основні етапи розвитку мов програмування.
3. Історія створення та особливості мови програмування Python.

Поняття «програма» та «мова програмування»

Комп'ютерна програма – набір інструкцій у вигляді слів, цифр, кодів, схем, символів, виражених у формі, придатній для зчитування комп'ютером, які приводять його у дію для досягнення певної мети або результату [1].

Крім того, поняття «комп'ютерна програма» визначають, як низку команд для комп'ютера, що є записом алгоритму однією з мов програмування [2].

Слід зазначити, що програма може бути записаною у текстовому вигляді на мовах програмування, збереженою на носіях інформації у вигляді файлу, поданою у графічному вигляді за допомогою блок-схем або занесеною до пам'яті обчислювальної системи у вигляді електричних сигналів.

Отже, «програму» можна уявити як набір послідовних команд (алгоритм) для об'єкта (виконавця), який повинен їх виконати для досягнення певної мети.

Наприклад, можна запрограмувати студента, надавши йому інструкцію «Як скласти екзамен по дисципліні «Спеціалізоване програмне забезпечення» на відмінно», яку він почне чітко виконувати. Слід зазначити, що дана інструкція (програма) для студента буде написаною на природній мові.

Однак, зазвичай, прийнято програмувати не людей, а обчислювальні машини, використовуючи при цьому спеціальні мови, це викликано тим, що машини не в змозі розуміти людську мову.

Встановлено, що мови програмування характеризуються [3]:

– синтаксичною однозначністю (не можна міняти місцями певні слова);

– синтаксичною обмеженістю (строго певний набір слів та символів).

Мова програмування – це система позначень для опису алгоритмів та структур даних.

Слід зазначити, що мову програмування визначає набір лексичних, синтаксичних та семантичних правил, які задають зовнішній вигляд програми, та дії, які виконує виконавець (комп'ютер) під її управлінням.

Таким чином, під «мовою програмування» слід розуміти штучну мову, створену для передачі команд комп'ютеру.

Основні етапи розвитку мов програмування

Перші програми писалися на машинній мові, зрозуміти їх було досить складно, крім того, навіть невелика програма складалася з безлічі рядків коду.

Ситуація ускладнювалася ще й тим, що кожна обчислювальна машина розуміла лише свою машинну мову. Людям, на відміну від машин, більш зрозумілі слова, ніж набори цифр. Прагнення людини оперувати словами, а не цифрами, призвело до появи *асемблерів* – мов, в яких замість чисельного позначення команд та областей пам'яті використовуються словесно-буквені.

Однак, з'явилася проблема, машина була не в змозі зрозуміти набори букв, виникла необхідність у певному перекладачі на машинну мову.

З часів появи асемблерів під кожну мову програмування створювалися *транслятори* – спеціальні програми, що перетворювали програмний код з мови програмування в машинний код. Слід зазначити, що механізм цього перекладу надзвичайно складний, виділяють *два основних способи трансляції* – компіляція програми або її інтерпретація [1].

Під час компіляції весь вихідний програмний код, що пише програміст, відразу переводиться у машинний код. Таким чином, створюється окремий виконуваний файл, який ніяк не пов'язаний з вихідним кодом, його запуск забезпечується операційною системою.

Під час інтерпретації виконання коду відбувається послідовно, рядок за рядком. Операційна система не зчитує вихідний код безпосередньо, а взаємодіє з інтерпретатором. Слід зазначити, що виконання компіляції відбувається швидше, адже вона є готовим машинним кодом, але на сучасних комп'ютерах зниження швидкості виконання під час інтерпретації, зазвичай, є не помітним.

Після асемблерів настав час мов програмування високого рівня. Для цих мов потрібно розробляти більш складні транслятори, тому що мови високого рівня зручніші для розуміння людиною, ніж для обчислювальної машини.

Крім того, на відміну від асемблерів, які залишаються прив'язаними до певних типів машин, мови високого рівня можуть переноситися (створивши програму, програміст має можливість запустити її на будь-якій машині).

Наступним значущим кроком стала поява об'єктно-орієнтованих мов програмування, за допомогою яких програміст наче управляє віртуальними об'єктами. Слід зазначити, що реалізація складних проектів здійснюється, як правило, за допомогою об'єктно-орієнтованого програмування.

Отже, комп'ютерні програми, якщо їх не подано у вигляді послідовності машинних кодів системи команд процесора обчислювальної системи, необхідно попередньо перетворити в такі коди за допомогою компілятора або виконати програму, використавши програмний інтерпретатор.

Історія створення та особливості мови програмування Python

Мову програмування Python створено приблизно в 1991 році голландцем Гвідо ван Россумом. Свою назву – «Пайтон» (або «Пітон») – мова отримала від назви телесеріалу «Повітряний цирк Монті Пайтона», а не класу плазунів. Після того, як Гвідо ван Россум розробив мову, він виклав її в мережу Інтернет, до її поліпшення приєдналося співтовариство провідних програмістів [3].

Мова програмування Python активно вдосконалюється і в даний час, постійно виходять нові версії. Офіційний сайт мови: http://python.org.

Особливості мови програмування Python – це інтерпретована мова програмування: вихідний код частинами перетворюється в машинний у процесі його читання спеціальною програмою – інтерпретатором.

Крім того, мова програмування Python має зрозумілий синтаксис, читати код досить легко, тому що він має мінімум допоміжних елементів (дужок, розділових знаків), а правила мови змушують програмістів робити необхідні відступи. Слід зазначити, що добре систематизований та оформлений текст з невеликою кількістю відволікаючих елементів читати і розуміти значно легше.

Мова програмування Python – це повноцінна (універсальна) мова програмування, яка підтримує об'єктно-орієнтоване програмування [4].

До особливостей мови програмування Python слід віднести те, що, якщо інтерпретатору Python дати команду import this (імпортувати «сам об'єкт»), то виведеться «*Дзен Пітона*», який показує ідеологію та особливості даної мови. Крім того, глибоке розуміння «дзену» приходить до тих, хто зможе освоїти мову Python у повній мірі та набуде досвіду практичного програмування, прикладами «Дзена Пітона» є [3]:

– Beautiful is better than ugly (красиве краще потворного);

– Explicit is better than implicit (явне краще неявного);

– Simple is better than complex (просте краще складного);

– Complex is better than complicated (складне краще ускладненого);

– Special cases aren't special enough to break the rules (винятки не настільки істотні, щоб порушувати правила);

– Errors should never pass silently (помилки ніколи не повинні замовчуватися);

– Now is better than never (зараз краще, ніж ніколи).

Слід зазначити, що інтерпретатор мови програмування Python виконує команди через посередника: пишемо рядок, натискаємо Enter, інтерпретатор виконує команди, отримуємо результат.

Дана можливість корисна для початківців, які тільки починають вивчати програмування та тестують якусь певну частину коду. Якщо ж працювати на компільованій мові, то необхідно спочатку написати код на вихідній мові програмування, потім скомпілювати та запустити виконуваний файл, щоб подивитися чи правильно працює код.

Крім того, якщо в процесі введення даних допущено помилку або потрібно повторити раніше використану команду, то, щоб не писати рядок спочатку, в консолі можна прокручувати список команд, використовуючи для цього стрілки вгору та вниз на клавіатурі. На відміну від консольного варіанту, у середовищі розробки IDLE мови програмування Python, де є інтерактивний режим роботи, можна спостерігати підсвічування синтаксису та прокручувати список раніше введених команд за допомогою комбінацій Alt+N, Alt+P.

Незважаючи на зручності інтерактивного режиму роботи, на практиці програмістам необхідно зберігати вихідний програмний код для подальшого використання, тому готуються файли, які потім передаються інтерпретатору на виконання. По відношенню до інтерпретованих мов програмування вихідний код часто називають *скриптом*.

Важливо: файл – це послідовність байтів, що має певну назву.

Частина назви файлу, яка слідує за останньою крапкою, називається *розширенням файлу* та вказує на тип файлу. Розширення файлу, зазвичай, надає програма, у якій його створено. Файли з кодом на мові програмування Python, зазвичай, мають розширення *py*.

Підготувати скрипти можна безпосередньо в середовищі розробки IDLE мови програмування Python: запустивши програму, в меню слід вибрати команду File – New File (Crtl+N), відкривши нове вікно, необхідно зберегти файл. Після написання коду слід знову зберегти файл. Запустити скрипт можна скориставшись командою меню Run – Run Module або клавішею F5, з’явиться результат виконання збереженого коду.

Важливо: якщо ввести програмний код, попередньо не зберігши файл, то підсвічування синтаксису буде відсутнім.

Слід зазначити, що скрипти можна готувати в будь-якому текстовому редакторі, головне, щоб він підтримував підсвічування синтаксису мови програмування Python.

1.2 Типи даних у мові програмування Python

План

1. Вбудовані типи даних мови програмування Python.
2. Цілочисельний тип даних та тип чисел з плаваючою точкою мови програмування Python.
3. Рядки у мові програмування Python.
4. Перетворення типів даних: функції int(), float(), str().
5. Логічний тип даних мови програмування Python.
6. Виведення даних з клавіатури у мові програмування Python.
7. Введення даних з клавіатури у мові програмування Python.
8. Створення списків у мові програмування Python.
9. Кортежі у мові програмування Python.
10. Словники у мові програмування Python.
11. Множини у мові програмування Python.
12. Складені структури даних у мові програмування Python.

Вбудовані типи даних мови програмування Python

Розглянемо такі типи даних мови програмування Python [1–5]:

– ***цілі числа*** (*integer*) – додатні та від'ємні цілі числа, а також 0;

– ***числа з плаваючою точкою*** (*float point*) – дробові числа;

– ***рядки*** (*string*) – набір символів, укладених в лапки.

Цілочисельний тип даних та тип чисел з плаваючою точкою мови програмування Python

Важливо: у мові програмування Python роздільником цілої та дробової частини є точка, а не кома, а лапки можуть бути одинарними або подвійними.

До різних типів даних мови програмування Python застосовують операції.

Операція – це виконання певних дій над даними (операндами) [6].

Для виконання конкретних дій потрібні спеціальні інструменти, які називають *операторами* (табл. 1.1).

Важливо: символ «+» по відношенню до чисел є операцією складання, по відношенню до рядків – *конкатенації* (з'єднання), а парний знак ** зводить перше число в ступінь другого.

Слід пам'ятати, що при складанні цілого числа та числа з плаваючою точкою, виходить число з плаваючою точкою, а якщо скласти будь-яке число та рядок, то інтерпретатор мови програмування Python видасть помилку.

Таблиця 1.1 – Математичні оператори та їх використання

Оператор	Операція	Приклад	Результат
+	Додавання	5 + 5	10
-	Віднімання	50 - 10	40
*	Множення	4 * 4	16
/	Десяткове ділення	7 / 2	3.5
//	Цілочисельне ділення	17 // 5	3
%	Остача від ділення	10 % 3	1
**	Піднесення до степеня	2 ** 4	16

Особливості створення змінних у мові програмування Python

З появою асемблерів під час звернення до даних використовують *змінні*. Механізм зв'язку між змінними та даними може відрізнятися у залежності від мови програмування та типів даних.

Важливо: дані зв'язуються з певним ім'ям, надалі звернення до даних можливо через це ім'я.

Зв'язок між даними та змінними встановлюється за допомогою знаку «=», така операція називається *присвоєння*. Наприклад, вираз *number* = 4 означає, що *number* – ім'я змінної, = – операція присвоєння, 4 – дані.

Важливо: у мові програмування Python вираз, що знаходиться справа від знаку присвоювання «=» обчислюється в першу чергу, цей результат обчислення запам'ятовується, потім зазначений результат обчислення присвоюється змінній, яка стоїть з лівої сторони від знаку присвоювання «=».

Слід зазначити, що імена змінних можуть бути довільними, але є декілька загальних правил їх написання [1]:

– необхідно надавати змінним осмислені імена, що будуть говорити про призначення даних, на які вони посилаються;

– ім'я змінної не повинно збігатися із командами мови, зарезервованими ключовими словами (табл. 1.2);

Таблиця 1.2 – Зарезервовані слова мови програмування Python

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	break

– ім’я змінної може містити букви у нижньому або верхньому регістрі, цифри від 0 до 9 або символ нижнього підкреслення.

Важливо: імена змінних не можуть починатися з цифри, не потрібно ставити на початку числа 0, наприклад, «05», бо це викличе помилку «некоректний символ».

Щоб дізнатися значення, на яке посилається змінна, перебуваючи в режимі інтерпретатора, необхідно викликати її (написати ім’я, натиснути Enter).

Математичні функції у мові програмування Python

Функції у програмуванні можна представити як ізольований блок коду, звернення до якого в процесі виконання програми може бути багаторазовим. Такі блоки потрібні, щоб скоротити обсяг вихідного коду: раціонально винести повторювані вирази в окремий блок і, у міру потреби, звертатися до нього.

Таким чином, достатньо оголосити функцію лише один раз і потім багаторазово використовувати код, який вона реалізує.

Мова програмування Python має математичні функції стандартної бібліотеки math. Щоб отримати до них доступ, необхідно ввести import math.

Прикладами функцій бібліотеки math є [7]:

– повернення абсолютного значення:

```
>>> math.fabs(-500.5)
500.5
```

– округлення вниз:

```
>>> math.floor(50.8)
50
```

– округлення вгору:

```
>>> math.ceil(50.2)
51
```

– обчислення факторіалу:

```
>>> math.factorial(8)
40320
```

– піднесення числа до степеня:

```
>>> math.pow(5, 3)
125.0
```

– обчислення кореня квадратного з числа:

```
>>> math.sqrt(625)
25.0
```

– перетворення значення в градусах у радіани:

```
>>> math.radians(180)
3.141592653589793
```

– перетворення значення в радіанах у градусну міру:

```
>>> math.degrees(math.pi)
180.0
```

У цій бібліотеці також присутні тригонометричні функції: sin(), cos(), tan(), asin(), acos(), atan().

Рядки у мові програмування Python

Рядок – це складний тип даних, що є послідовністю символів.

Рядки в мові програмування Python можуть міститися як в одиночних, так і в подвійних лапках. Однак, початок і кінець рядка повинен відкриватися та закриватися однаковим типом лапок. Метою використання двох видів лапок є можливість створювати рядки, що містять лапки чи апострофи. Отже, усередині одинарних лапок можна розташувати подвійні та навпаки.

Важливо: можна також використовувати три одинарні або три подвійні лапки, наприклад, щоб створити багаторядкові рядки (текст вірша).

Усередині потрійних лапок застосовують символи переходу на новий рядок (\n), слід зазначити, що пропуски у такому випадку зберігаються.

Для рядків існують операції *конкатенації* (+) і *дублювання* (*).

Функція format() у мові програмування Python

Розглянемо, як розміщувати різноманітні значення всередині рядків, застосовуючи різні формати, цю можливість можна використовувати для створення певного зовнішнього вигляду під час виведення інформації.

Наприклад, визначимо кілька змінних:

```
>>> n = 30 # цілочисельну
>>> f = 10.5 # число з плаваючою точкою
>>> s = ‘hello’ # рядок
```

За допомогою функції format() можна певним чином розмістити ці значення в одному рядку у тому порядку, як вони вказані для функції:

```
>>> '{} {} {}'.format(n, f, s)
'30 10.5 hello'
```

Порядок виведення значень в рядку можна вказувати у фігурних дужках таким чином (нумерація починається з нуля):

```
>>> '{2} {0} {1}'.format(n, f, s)
'hello 30 10.5'
```

Значення виводяться на екран з форматуванням за замовчуванням. Python може вказувати специфікатори типу після символу «:» для форматування.

Слід зазначити відомі специфікатори типів: s – рядок, d – ціле число в десятковій системі числення, f – число з плаваючою точкою у десятиковій системі числення.

Отже, якщо потрібно вивести число з плаваючою точкою з точністю до трьох знаків, то можна використати запис «.3» перед ім'ям специфікатора f:

```
>>> '{0:d} {1:.3f} {2:s}'.format(n, f, s)
'30 10.500 hello'
```

Керуючі символи у мові програмування Python

Слід зазначити, що мова програмування Python дозволяє створювати керуючі послідовності всередині рядків. Найбільш поширена послідовність \n, яка означає перехід на новий рядок, так можна створити багаторядкові рядки з однорядкових. Також можна у мові програмування Python застосовувати послідовність \t (табуляція), яка використовується для вирівнювання тексту, завдяки відступам. Послідовності \' або \'' використовуються, щоб помістити в рядок одинарні або подвійні лапки, що оточені таким ж лапками. Якщо потрібно ввести зворотний слеш (\), перед ним необхідно поставити ще один \\.

Конкатенація рядків у мові програмування Python

У мові програмування Python за допомогою оператора + можна об'єднати рядки або рядкові змінні. Наприклад, ім'я та прізвище зберігаються в різних змінних і їх необхідно об'єднати для виведення повного імені. Такий спосіб об'єднання рядків називається *конкатенацією* [8].

Важливо: під час конкатенації рядків мова програмування Python не додає пропуски, їх потрібно явно додавати.

Дублювання рядків у мові програмування Python

Слід зазначити, що мова програмування Python має оператор * (зірочка), який можна використовувати для того, щоб продублювати рядок.

Доступ до елементів рядка по індексу у мові програмування Python

У послідовності важливий порядок символів, у кожного символу в рядку є унікальний порядковий номер – *індекс*. Можна звертатися до конкретного символу в рядку та витягувати його за допомогою оператора *індексування*, який записується за допомогою квадратних дужок з номером символу в них.

Наприклад, вираз ‘abetka’[1] призводить до вилучення другого символу. Справа в тому, що індексація починається не з одиниці, а з нуля.

Коли потрібно витягти перший символ, то оператор індексування повинен виглядати так: [0]. Також дозволено отримувати символи, починаючи відлік з кінця, у цьому випадку відлік починається з -1 (останній символ).

Очевидно, що зручніше працювати не з самими рядками, а зі змінними, які на них посилаються, тоді результат виконання виразу індексування можна присвоїти іншій змінній. Щоб отримати один символ рядка, необхідно задати індекс цього символу в рядку всередині квадратних дужок після імені рядка:

```
>>> abetka = ‘abcdefghijklmnopqrstuvwxyz’
>>> abetka[0]
‘a’
>>> abetka[6]
‘g’
>>> abetka[-1]
‘z’
```

Важливо: індекси мають значення в діапазоні від 0 до довжини рядка.

Функція slice[start: end: step] – зрізи у мові програмування Python

Слід зазначити, що можна витягати з рядка не один символ, а декілька, тобто отримувати зріз (підрядок) [9]. Оператор вилучення зрізу з рядка виглядає так: [X: Y], X – це індекс початку зрізу, а Y – його закінчення (причому символ з номером Y у зріз вже не входить). Якщо відсутній перший індекс, то зріз береться від початку до другого індексу, при відсутності другого індексу, зріз береться від першого індексу до кінця рядка.

Крім того, можна витягати символи не підряд, а через певну кількість символів, оператор індексування виглядає так: [X: Y: Z], де Z – це крок, через який здійснюється вибір елементів.

Отже, з рядка можна «витягти» не лише один символ, але і підрядок за допомогою функції slice. У квадратних дужках запису цієї функції вказується:

– індекс початку підрядка start;

– індекс кінця підрядка end;

– розміру кроку step.

Крім того, деякими з цих параметрів можна знехтувати.

Можна у підрядок включити символи розташовані, починаючи з символу, на який вказує індекс start, і закінчити символом, на який вказує індекс end.

Наприклад, дано рядок, необхідно:

– вирізати послідовність символів від початку і до кінця:

```
>>> abetka = ‘abcdefghijklmnopqrstuvwxyz’
>>> abetka[:]
‘abcdefghijklmnopqrstuvwxyz’
```

– вирізати послідовність символів від 20-го символу та до кінця:

```
>>> abetka = ‘abcdefghijklmnopqrstuvwxyz’
>>> abetka[20:]
‘uvwxyz’
```

– вирізати послідовність символів від 12-го символу по 15-й символ, що не включається:

```
>>> abetka = ‘abcdefghijklmnopqrstuvwxyz’
>>> abetka[12:15]
‘mno’
```

– вирізати послідовність останніх 5 символів:

```
>>> abetka = ‘abcdefghijklmnopqrstuvwxyz’
>>> abetka[-5:]
‘vwxyz’
```

– вирізати послідовність символів від 18-го символу та до 3-го з кінця, який не включається:

```
>>> abetka = ‘abcdefghijklmnopqrstuvwxyz’
>>> abetka[18:-3]
‘stuvw’
```

– вирізати послідовність символів від 6-го символу з кінця та до 2-го символу з кінця, який не включається:

```
>>> abetka = ‘abcdefghijklmnopqrstuvwxyz’
>>> abetka[-6:-2]
‘uvwx’
```

– вирізати кожний 7-й символ з початку та до кінця:

```
>>> abetka = ‘abcdefghijklmnopqrstuvwxyz’
>>> abetka[::7]
‘ahov’
```

– вирізати кожний 3-й символ від 4-го символу з початку та до 20-го символу, який не включається:

```
>>> abetka = ‘abcdefghijklmnopqrstuvwxyz’
>>> abetka[4:20:3]
‘ehknqt’
```

– вирізати кожний 4-й символ, починаючи від 19-го символу та до кінця:

```
>>> abetka = ‘abcdefghijklmnopqrstuvwxyz’
>>> abetka[19::4]
‘tx’
```

– вирізати кожний 5-й символ, починаючи з початку та до 21-го символу, який не включається:

```
>>> abetka = ‘abcdefghijklmnopqrstuvwxyz’
>>> abetka[:21:5]
‘afkpu’
```

– перевернути рядок з кінця на початок:

```
>>> abetka = ‘abcdefghijklmnopqrstuvwxyz’
>>> abetka[::-1]
‘zyxwvutsrqponmlkjihgfedcba’
```

Слід зазначити, що запис неправильних початкового та кінцевого індексів дає порожній рядок.

Функція len() – довжина рядка у мові програмування Python

У мові програмування Python існує спеціальна функція len (), що дозволяє виміряти довжину рядка, результатом виконання якої є число, що показує кількість символів в рядку [10].

Приклад того, як функція len() підраховує символи в рядку:

```
>>> abetka = ‘abcdefghijklmnopqrstuvwxyz’
>>> len(abetka)
26
```

Важливо: функцію len() можна використовувати і для інших типів послідовностей.

Функція split() – розділення рядка у мові програмування Python

Функція split() розбиває рядок на окремі рядки та розміщує їх у списку.

Використовуючи функцію split(), необхідно вказати символ-розділювач:

```
>>> abetka = ‘abcdef, ghijklm, nopqrs, tuvwxyz’
>>> abetka.split(‘,’)
[‘abcdef’, ‘ghijklm’, ‘nopqrs’, ‘tuvwxyz’]
```

Якщо не вказати символ-розділювач, то функція split() використає будь-яку послідовність пропусків, а також символи нового рядка та табуляцію:

```
>>> abetka = ‘abcdef, ghijklm, nopqrs, tuvwxyz’
>>> abetka.split()
[‘abcdef,’, ‘ghijklm,’, ‘nopqrs,’, ‘tuvwxyz’]
```

Важливо: для виклику функції split(), навіть без аргументів, потрібно додавати круглі дужки.

Функція join() – об’єднання рядків у мові програмування Python

Слід зазначити, що функція join() є протилежністю функції split().

Важливо: функція join() об’єднує список рядків в один рядок.

Виклик функції виглядає так: необхідно вказати рядок, що об’єднує інші, а потім – список рядків для об’єднання: рядок.join(список рядків).

Функція replace() – заміна символів у мові програмування Python

Зміна одного підрядка на інший виконується за допомогою функції replace(). Для цього необхідно передати у цю функцію «старий підрядок» (який треба замінити), «новий підрядок» (яким треба замінити) і кількість входжень старого підрядка: рядок.replace(старий рядок, новий підрядок, кількість замін).

Важливо: якщо не записувати останній аргумент (кількість замін), буде проведена заміна усіх входжень.

Перетворення типів даних: функції int(), float(), str()

Бувають випадки, коли програма отримує дані у вигляді рядків, а оперувати повинна числами (або навпаки), тоді використовуються *спеціальні функції* (особливі оператори), що дозволяють перетворити один тип даних в інший (табл. 1.3) [2].

Таблиця 1.3 – Перетворення типів даних

Вираз	Результат виконання
int (“62”)	62
int (5.04)	5
int (“comp 486”)	Помилка
float (62)	62.0
float (“62”)	62.0
str (62)	‘62’
str (5.04)	‘5.04’

Наприклад, функція int() перетворює переданий їй рядок (або число з плаваючою точкою) в ціле, функція float() – у число з плаваючою точкою (дробове число), функція str() перетворює переданий їй аргумент у рядок:

```
>>> int(12.5)
12
>>> float(50)
50.0
>>> str(100)
‘100’
```

Логічний тип даних мови програмування Python

У реальному житті часто доводиться погоджуватися або заперечувати твердження, подію чи факт, наприклад, «сума чисел 3 та 5 більша, ніж 7» є правдивим твердженням, а «сума чисел 3 та 5 менша, ніж 7» – хибним.

З точки зору логіки дані твердження допускають лише два результати: «так» (істина) і «ні» (хибне). Подібне використовується і у програмуванні: якщо результатом обчислення виразу може бути лише істина або хибне, то такий вислів називається *логічним (булевим)*, тому у *логічному (булевому) типі даних* всього два можливих значення: True (істина) або 1 та False (хибне) або 0, крім того, ці значення можуть бути результатом логічних виразів [11].

Важливо: булеві значення True та False у мові програмування Python записуються без лапок, їх назви починаються з великих літер T та F, тоді як інша частина слова пишеться маленькими буквами.

Слід зазначити, що у мові програмування Python використовують логічні оператори (спеціальні знаки), а саме [12]:

– > (більше, ніж);

– < (менше, ніж);

– >= (більше або дорівнює);

– <= (менше або дорівнює);

– == (два знаки «дорівнює» означає рівність);

– != (не дорівнює).

Важливо: операція присвоєння позначається одним знаком «дорівнює», а операція рівності – двома знаками «дорівнює», у мові програмування Python це абсолютно різні операції.

Однак, на практиці використовують складні логічні вирази, у даному випадку застосовують спеціальні логічні (булеві) оператори *and* та *or*, що об'єднують два та більше простих логічних виразів.

Щоб отримати True (істину) під час використання оператора *and*, необхідно, щоб результати обох простих виразів, які пов'язує цей оператор, були істинними. Якщо хоча б в одному випадку результатом виявиться False (хибне), то і весь складний вираз є хибним.

Щоб отримати True (істину) під час використання оператора *or*, необхідно, щоб результат хоча б одного простого виразу, що входить до складу складного, був істинним. Під час роботи з оператором or складний вираз стає хибним лише тоді, коли хибні всі його прості вирази.

Виведення даних з клавіатури у мові програмування Python

Відомо, що комп'ютерні програми обробляють дані за допомогою певних операцій, що визначив програміст та які обумовлені поставленими завданнями.

Дані у програму можна «закласти» у процесі її розробки, однак, на практиці програма повинна обробляти різні дані, які надходять до неї із зовнішніх джерел, якими можуть виступати *файли* або *клавіатура*.

Функція print() у мові програмування Python

Коли інформація вводиться із клавіатури, а результати виводяться на екран монітора, то можна говорити про інтерактивний режим роботи програми. Програма, обмінюючись інформацією із зовнішнім для неї середовищем, може виводити та отримувати дані в процесі виконання. Виведенням на екран у мові програмування Python займається функція print().

Функція print() виводить на екран вміст, що записаний у лапках (але без них), додаючи пропуск між кожним виведеним елементом, а також символ переходу на новий рядок (\n) у кінці [1]:

```
>>> print('Discipline', 'specialized', 'software.')
Discipline specialized software.
```

Важливо: функція print() має необов'язкові параметри end і sep, за допомогою яких можна вказати відповідно текст, який повинен бути в кінці виведення функції, і текст, який повинен бути розділювачем усередині вмісту функції.

Щоб змінити розділювач (за замовчуванням пропуск) між елементами, які виводить функція print(), треба використати її необов'язковий параметр sep і вказати розділювач.

Наприклад:

```
>>> print('Discipline', 'specialized', 'software.', sep=',')
Discipline,specialized,software.
```

Слід зазначити, що замість символу нового рядка (\n) можна використовувати інший рядок, вказаний з допомогою параметра end.

Цей прийом застосовують у тих випадках, коли необхідно відмінити символ нового рядка (\n), який автоматично додається в кінці кожного виведення за допомогою функції print().

Введення даних з клавіатури у мові програмування Python

Для виконання ряду програм, зазвичай, потрібна деяка інформація, яку повинен ввести користувач.

Функція input() у мові програмування Python

Введення даних з клавіатури, починаючи з програмної версії Python 3.0, здійснюється за допомогою функції input(). Якщо функція виконується, то потік виконання програми зупиняється в очікуванні даних, які користувач повинен ввести за допомогою клавіатури. Після введення даних та натискання Enter, функція input() завершує своє виконання та повертає результат, який є рядком символів, введених користувачем.

Розглянемо приклад введення даних з клавіатури в програму [2]:

```
>>> input()
37
'37'
```

```
>>> input()
Let’s start the modular testing!
“Let’s start the modular testing!”
>>>
```

Коли програма пропонує користувачеві що-небудь ввести, то останній може не зрозуміти, введення яких даних від нього очікують, тому необхідно якимось чином це уточнити.

Функція input() може приймати необов’язковий аргумент-запрошення строкового типу, під час її виконання повідомлення буде з’являтися на екрані та інформувати користувача про необхідні дані [12]:

```
>>> input(“Введіть назву дисципліни: ”)
Введіть назву дисципліни: Спеціалізоване програмне забезпечення
‘Спеціалізоване програмне забезпечення’
>>> input(“Введіть ім’я викладача: ”)
Введіть ім’я викладача: Творошенко Ірина Сергіївна
‘Творошенко Ірина Сергіївна’
>>> input(“Введіть бажану кількість балів за екзамен: ”)
Введіть бажану кількість балів за екзамен: 30
‘30’
>>>
```

У ранніх версіях мови програмування Python існувало дві вбудовані функції, що дозволяли отримувати дані з клавіатури [13]:

– raw_input(), яка повертає в програму рядок;

– input(), яка повертає в програму число.

Починаючи з програмної версії Python 3.0, якщо необхідно отримати число, то результат виконання функції input() змінюють за допомогою функцій int() або float().

Результат, що повертається функцією input(), зазвичай, присвоюється змінній для подальшого використання у програмі.

Створення списків у мові програмування Python

Списки дозволяють зберігати в одному місці множину взаємопов’язаних даних. Слід зазначити, що робота зі списками відноситься до числа найбільш видатних можливостей мови програмування Python. Важливо, що у список можна додати нові елементи, видалити або перезаписати існуючі. Крім того, одне і те ж значення може зустрічатися у списку кілька разів.

Отже, список можна створити для зберігання букв алфавіту, цифр від 0 до 9 або рядків, наприклад, власників всіх земельних ділянок Харківської області. У списку можна зберігати будь-яку інформацію, причому дані в списку не зобов'язані бути якимось чином пов'язані один з одним. Враховуючи, що список, зазвичай, містить більше одного елемента, то рекомендується надавати спискам імена у множині: letters, digits, names і т. д.

Таким чином, списки – це послідовності, що можуть змінюватися.

Списки у мові програмування Python, як і рядки, є упорядкованими послідовностями. Однак, на відміну від рядків, списки складаються не з символів, а з різних об'єктів (значень, даних), і беруться не в лапки, а в квадратні дужки []. Об'єкти в списках відокремлюються один від одного за допомогою коми [14].

Отже, списки можуть складатися з різних об'єктів: чисел, рядків і навіть інших списків, в останньому випадку списки називають *вкладеними*.

Розглянемо приклади списків:

– [20, 30, -20, 60, -40] – список цілих чисел;

– [4.15, 10.95, 16.45, 9.30, 10.0, 11.5] – список із дробових чисел;

– ['Katy', 'Sergei', 'Oleg', 'Dasha'] – список із рядків;

– ['Москва', 'Титова', 10, 150] – змішаний список;

– [[0, 0, 0], [0, 0, 1], [0, 1, 0]] – список, який складається із списків.

Важливо: списки у мові програмування Python можна порівняти із масивами у інших мовах програмування.

Як і над рядками над списками можна виконувати операції з'єднання та повторення:

```
>>> [45, -10, 'травень'] + [20, 48.0, 35]
[45, -10, 'травень', 20, 48.0, 35]
>>> [[0,0],[0,1],[1,1]] * 2
[[0, 0], [0, 1], [1, 1], [0, 0], [0, 1], [1, 1]]
```

На відміну від рядків, списки – це послідовності, що можуть змінюватися. Якщо уявити рядок як об'єкт у пам'яті і коли над ним виконуються операції конкатенації та повторення, то цей рядок не змінюється, а в результаті операції створюється інший рядок в іншому місці пам'яті.

У рядок можна додати новий символ або видалити існуючий, не створюючи при цьому нового рядка. Зі списком інша справа. Під час виконання операцій інші списки можуть не створюватися, а змінюється безпосередньо оригінал.

Зі списків можна видаляти елементи, додавати нові, при цьому слід пам’ятати, багато що залежить від того, як оперують змінними. Бувають ситуації, коли списки все-таки копіюються. Наприклад, результат операції присвоюється інший змінній. Символ у рядку змінити не можна, а елемент списку – можна, у списку можна замінити цілий зріз.

Важливо: якщо присвоїти один список більше, ніж одній змінній, то зміни у списку в одному місці спричинять за собою його зміни в інших місцях.

Отже, значення списку можна скопіювати в незалежний новий список за допомогою одного зі способів:

– функції copy();

– функції list();

– розділенням списку за допомогою [:].

Наприклад, список присвоєно змінній *a*, списки b, c, d – це копії списку *a*:

```
>>> a = [1, 2, 3]
>>> b = a.copy()
>>> c = list(a)
>>> d = a[:]
```

Слід зазначити, що b, c, d – це нові об’єкти, що мають свої значення, які не пов’язані з оригінальним списком елементів [1, 2, 3], на який посилається змінна *a*. Крім того, зміни в *a* не впливають на копії b, c, d.

Важливо: під час створення копій списку, використовуючи функції copy() та list() або конструкції [:], зміни в оригінальному списку не впливають на зміни у списках-копіях, так як списки-копії є вже новими об’єктами.

Доступ до елементів списку та довжина списку

Враховуючи, що списки є упорядкованими наборами даних, то для доступу до будь-якого елементу списку слід повідомити Python позицію (індекс) потрібного елемента. Індекси приймають тільки цілочисельні значення. Щоб звернутися до елементу у списку, необхідно вказати ім’я списку, а потім індекс елемента в квадратних дужках.

Слід пам’ятати, що індекси починаються з 0, а не з 1. Цей принцип зустрічається у більшості мов програмування, у мові програмування Python він пояснюється особливостями низько рівневої реалізація операцій зі списками.

Отже, перший елемент списку відповідає індексу 0. Другий елемент списку відповідає індексу 1 і т. д. Наприклад, щоб звернутися до четвертого елементу списку, слід вказати елемент з індексом 3. Таким чином, як і для рядків, зі списку можна отримати конкретне значення, вказавши його індекс.

Крім того, функція len() повертає кількість елементів списку.

Отже, можна отримувати доступ до об'єктів списку по їх індексам, витягувати зрізи, вимірювати довжину списку:

```
>>> abetka = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> len(abetka)
7
>>> abetka[0]
'a'
>>> abetka[4]
'e'
>>> abetka[0:3]
['a', 'b', 'c']
>>> abetka[3:]
['d', 'e', 'f', 'g']
```

Якщо необхідно дізнатися індекс елемента у списку за його значенням, використовується функція index():

```
>>> abetka = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> abetka.index('c')
2
```

Функції, що застосовують до списків у мові програмування Python

Функція list() у мові програмування Python перетворює інші типи даних у список, наприклад, рядок перетворюється у список:

```
>>> list('sun')
['s', 'u', 'n']
```

Щоб перетворити рядок у список можна також скористатися функцією split(), вказавши рядок-розділювач:

```
>>> data = '12/5/1980'
>>> data.split('/')
['12', '5', '1980']
```

Функція split() перетворює рядок у список елементів, розділених іншим рядком-розділювачем.

Для перетворення списку у рядок можна скористатися функцією join():

```
>>> abetka = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> ', '.join(abetka)
'a', 'b', 'c', 'd', 'e', 'f', 'g'
```

На відміну від рядків, список можна змінити:

```
>>> abetka = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]
>>> abetka[5] = ‘k’
>>> abetka
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘k’, ‘g’]
```

Додавання елементів у список забезпечується викликом функції append():

```
>>> abetka = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]
>>> abetka.append(’h’)
>>> abetka
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’]
```

Функція append() додає елементи лише в кінець списку, а коли потрібно додати елемент у конкретну позицію, то використовується функція insert():

```
>>> abetka = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]
>>> abetka.insert(3, ’h’)
>>> abetka
[‘a’, ‘b’, ‘c’, ‘h’, ‘d’, ‘e’, ‘f’, ‘g’]
```

Важливо: якщо вказати позицію 0, то елемент буде додано в початок списку, якщо позиція знаходиться за межами списку, то елемент буде додано в кінець списку.

Для об’єднання одного списку з іншим використовують функцію extend():

```
>>> abetka1 = [‘a’, ‘b’, ‘c’]
>>> abetka2 = [‘d’, ‘e’, ‘f’, ‘g’]
>>> abetka1.extend(abetka2)
>>> abetka1
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]
```

Крім функції extend(), можна також використовувати операцію += для об’єднання списків:

```
>>> abetka1 = [‘a’, ‘b’, ‘c’]
>>> abetka2 = [‘d’, ‘e’, ‘f’, ‘g’]
>>> abetka1 += abetka2
>>> abetka1
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]
```

Слід зазначити, якщо використовувати функцію append(), то список abetka2 було б додано як один елемент списку, замість того, щоб об’єднати його елементи зі списком abetka1:

```
>>> abetka1 = [‘a’, ‘b’, ‘c’]
>>> abetka2 = [‘d’, ‘e’, ‘f’, ‘g’]
>>> abetka1.append(abetka2)
>>> abetka1
[‘a’, ‘b’, ‘c’, [‘d’, ‘e’, ‘f’, ‘g’]]
```

Отже, список може містити елементи різних типів.

Для видалення елементів зі списку користуються функцією del():

```
>>> abetka = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]
>>> del abetka[-1]
>>> abetka
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]
```

Важливо: якщо видаляється заданий елемент, то всі інші елементи, які йдуть слідом за ним, зміщуються вліво, щоб зайняти місце видаленого елемента, а довжина списку зменшується на одиницю.

Слід зазначити, що зі списку можна видалити елемент не лише за індексом, а і за значенням, використовуючи функцію remove():

```
>>> abetka = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]
>>> abetka.remove(‘g’)
>>> abetka
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]
```

Крім того, можна видалити елемент зі списку і, водночас, отримати його за допомогою функції pop(). Якщо викликати функцію pop() та вказати деякий зсув, то вона поверне елемент, що знаходиться в заданій позиції, якщо аргумент не вказано, то буде використано значення -1:

```
>>> abetka = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]
>>> abetka.pop()
‘g’
>>> abetka
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]
```

У мові програмування Python наявність елемента у списку перевіряється за допомогою оператора in.

```
>>> abetka = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]
>>> ‘a’ in abetka
True
>>> ‘k’ in abetka
False
```

Для перевірки щодо відсутність елемента у списку користуються поєднанням операторів not in:

```
>>> abetka = [‘a’, ’b’, ’c’, ’d’, ’e’, ’f’, ’g’]
>>> ‘a’ not in abetka
False
>>> ‘k’ not in abetka
True
```

Щоб визначити, скільки разів певне значення зустрічається у списку, використовується функція count():

```
>>> abetka = [‘a’, ’b’, ’c’, ’d’, ’e’, ’f’, ’g’]
>>> abetka.count(’c’)
1
```

Для сортування (зміни порядку розташування елементів) списку за їхнім значенням у мові програмування Python є дві функції [1]:

– sort() – сортує сам список (змінює його);

– sorted() – повертає відсортовану копію списку (без зміни оригінального списку).

Важливо: якщо елементи списку є числами, то вони за замовчуванням сортуються за зростанням, якщо – рядками, то сортуються в алфавітному порядку:

```
>>> numbers = [5, 3, 7.0, 4]
>>> numbers.sort()
>>> numbers
[3, 4, 5, 7.0]
```

За замовчуванням, список сортується за зростанням, але додавши аргумент зі значенням reverse=True, список відсортується за спаданням:

```
>>> numbers = [5, 3, 7.0, 4]
>>> numbers.sort(reverse=True)
>>> numbers
[7.0, 5, 4, 3]
```

Слід зазначити, що для того, щоб переставити елементи списку у зворотному порядку, використовується функція reverse():

```
>>> abetka = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]
>>> abetka
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]
>>> abetka.reverse()
>>> abetka
[‘g’, ‘f’, ‘e’, ‘d’, ‘c’, ‘b’, ‘a’]
```

Слід зазначити, що необхідність зберігання наборів чисел виникає у програмах по багатьом причинам. Наприклад, у програмах обчислювального характеру завжди працюють з наборами чисел: кадастрові номери, координати просторових об’єктів, відстань, чисельність населення і т. д.

Саме списки ідеально підходять для зберігання наборів чисел, а мова програмування Python надає спеціальні засоби для ефективної роботи з числовими списками, навіть якщо список містить множину елементів.

Для спрощення побудови числових послідовностей використовують функцію range():

```
>>> numbers = list(range(1,6))
>>> numbers
[1, 2, 3, 4, 5]
```

Важливо: функція range() дозволяє генерувати числові послідовності у заданому діапазоні.

Для побудови списку парних чисел від 1 до 10:

```
>>> numbers = list(range(2,11,2))
>>> numbers
[2, 4, 6, 8, 10]
```

Крім того, деякі вбудовані функції мови програмування Python призначені для роботи з числовими списками. Наприклад, можна легко дізнатися мінімум, максимум і суму елементів числового списку:

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(numbers)
0
>>> max(numbers)
9
>>> sum(numbers)
45
```

Кортежі у мові програмування Python

Кортежі, як і списки, є послідовностями будь-яких елементів. На відміну від списків кортежі є *незмінними*, це означає, що не можна додати, видалити або змінити елементи кортежу після того, як його створено (не можна знищити елементи кортежу помилково) [1].

Для створення порожнього кортежу використовується оператор ():

```
>>> numbers = ()
>>> numbers
()
```

Щоб створити кортеж, що містить один елемент або більше, необхідно поставити після кожного елемента кому, крім останнього:

```
>>> abetka = ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’
>>> abetka
(‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’)
```

Кортежі дозволяють присвоїти значення для кількох змінних за один раз:

```
>>> abetka = (‘a’, ‘b’, ‘c’)
>>> k, m, n = abetka
>>> k
‘a’
>>> m
‘b’
>>> n
‘c’
```

Крім того, існує можливість використовувати кортежі, щоб обмінюватися значеннями за допомогою одного виразу, без застосування тимчасової змінної:

```
>>> abetka1 = (‘abc’)
>>> abetka2 = (‘defg’)
>>> abetka1, abetka2 = abetka2, abetka1
>>> abetka1
‘defg’
>>> abetka2
‘abc’
```

Слід зазначити, що функція перетворення tuple() створює кортежі:

```
>>> abetka1 = [‘a’, ‘b’, ‘c’]
>>> tuple(abetka1)
(‘a’, ‘b’, ‘c’)
```

Словники у мові програмування Python

Словники – структури даних, призначені для об'єднання взаємозалежної інформації. Слід зазначити, що словники дозволяють моделювати різноманітні реальні просторові об'єкти.

Наприклад, можна створити словник, який описує певний тематичний шар, і зберегти у ньому скільки завгодно інформації про усі просторові об'єкти даного шару. Це може бути назва вулиці, номер будинку, площа, координати розташування об'єкта та будь-які інші атрибути.

Як правило, у словниках зберігаються будь-які два види інформації, здатні утворити пари, наприклад, список слів та їх значень, список власників та кадастрові номера їх земельних ділянок, список тематичних шарів та просторові об'єкти, що на них знаходяться і т. д.

Словник дуже схожий на список, але порядок елементів в ньому не має значення, а елементи вибираються за допомогою *унікального ключа*, що відповідає певному значенню словника.

Таким ключем в основному служить рядок, але він може бути об'єктом одного із незмінних типів: булевим значенням, цілим числом, числом з плаваючою точкою, кортежем тощо. Словники можна змінювати, це означає, що можна додати, видалити та змінити їх елементи, які мають вигляд ключ-значення. Якщо ключ визначено, то змінити його не можна.

Важливо: в інших мовах програмування словники можуть називатися «асоціативними масивами», «хешами» або «хеш-таблицею».

У мові програмування Python словник також називається dict.

Введення в словники у мові програмування Python

Таким чином, одним із складних типів даних (крім рядків та списків) у мові програмування Python є словники.

Словник – це змінний (як список) неупорядкований (на відміну від рядків та списків) набір пар «ключ: значення» [2].

Найпростішим словником є порожній словник, що не містить ні ключів, ні значень, і створюється за допомогою оператора {}:

```
>>> dict = {}
>>> dict
{}
```

Крім того, щоб створити словник зі значеннями, потрібно помістити у фігурні дужки {} пари «ключ: значення», розділені комами.

>>> dict = {‘name’: ‘Ukraine’, ‘area’: ‘603 628’}
>>> dict
{‘name’: ‘Ukraine’, ‘area’: ‘603 628’}

Щоб уявити словник, можна провести аналогію зі звичайним словником, наприклад, англійсько-українським. Кожне англійське слово у такому словнику має українське слово-переклад: cat – кішка, dog – собака, table – стіл і т. д.

Якщо англійсько-український словник описувати за допомогою мови програмування Python, то англійські слова будуть ключами, а українські – їх значеннями: {‘cat’: ‘кішка’, ‘dog’: ‘собака’, ‘bird’: ‘пташка’, ‘mouse’: ‘миша’}.

Важливо: саме за допомогою фігурних дужок визначається словник.

Таким чином, синтаксис словника на мові програмування Python можна описати такою схемою: {ключ:значення, ключ:значення, ключ:значення, …}.

Якщо створити словник у інтерпретаторі мови програмування Python, то після натискання Enter можна спостерігати, що послідовність виведення пар «ключ: значення» не збігається з тим, що було введено:

>>> {‘ГКЗ 2017-1’: ‘26’, ‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2016-1’: ‘19’, ‘ГКЗ 2015-1’: ‘15’, ‘ГКЗ 2015-2’: ‘21’, ‘ГКЗ 2014-1’: ‘16’, ‘ГКЗ 2014-2’: ‘17’, ‘ГКЗ 2014-3’: ‘22’, ‘М ГКЗ 2017-1’: ‘15’, ‘М ГКЗ 2017-2’: ‘15’}

{‘М ГКЗ 2017-1’: ‘15’, ‘М ГКЗ 2017-2’: ‘15’, ‘ГКЗ 2016-1’: ‘19’, ‘ГКЗ 2017-1’: ‘26’, ‘ГКЗ 2014-1’: ‘16’, ‘ГКЗ 2014-2’: ‘17’, ‘ГКЗ 2014-3’: ‘22’, ‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2015-2’: ‘21’, ‘ГКЗ 2015-1’: ‘15’}

Функції, що застосовують до словників у мові програмування Python

Словники, як і списки, є змінним типом даних: можна змінювати, додавати та видаляти елементи (пари «ключ: значення»). Спочатку словник необхідно створити порожнім (наприклад, d = {}), а потім заповнити його елементами.

Отже, функція dict() використовується для перетворення послідовності з двох значень у словник. Перший елемент кожної послідовності застосовується як ключ, а другий – як значення.

Слід зазначити, що додати елемент у словник досить легко, потрібно просто звернутися до елементу по його ключу та присвоїти йому значення.

Якщо ключ вже існує у словнику, то поточне значення буде замінено новим. Якщо ключ новий, то він і вказане значення будуть додані у словник.

Таким чином, додавання та зміна елемента у словнику має однаковий синтаксис – словник[ключ] = значення.

>>> groups = {‘ГКЗ 2017-1’: ‘26’, ‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2016-1’: ‘19’, ‘ГКЗ 2015-1’: ‘15’, ‘ГКЗ 2015-2’: ‘21’, ‘ГКЗ 2014-1’: ‘16’, ‘ГКЗ 2014-2’: ‘17’, ‘ГКЗ 2014-3’: ‘22’, ‘М ГКЗ 2017-1’: ‘15’, ‘М ГКЗ 2017-2’: ‘15’}

>>> groups[‘ГКЗ 2018-1’] = ‘22’

>>> groups[‘М ГКЗ 2017-1’] = ‘16’

>>> groups

{‘М ГКЗ 2017-1’: ‘16’, ‘М ГКЗ 2017-2’: ‘15’, ‘ГКЗ 2016-1’: ‘19’, ‘ГКЗ 2017-1’: ‘26’, ‘ГКЗ 2014-1’: ‘16’, ‘ГКЗ 2018-1’: ‘22’, ‘ГКЗ 2014-2’: ‘17’, ‘ГКЗ 2014-3’: ‘22’, ‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2015-2’: ‘21’, ‘ГКЗ 2015-1’: ‘15’}

Використовуючи функцію update(), можна скопіювати ключі і значення з одного словника в інший:

>>> groups1 = {‘ГКЗ 2017-1’: ‘26’, ‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2016-1’: ‘19’}

>>> groups2 = {‘ГКЗ 2015-1’: ‘15’, ‘ГКЗ 2015-2’: ‘21’}

>>> groups1.update(groups2)

>>> groups1

{‘ГКЗ 2017-1’: ‘26’, ‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2016-1’: ‘19’, ‘ГКЗ 2015-1’: ‘15’, ‘ГКЗ 2015-2’: ‘21’}

Важливо: якщо в другому словнику будуть знаходитися такі ж ключі, що і в першому, то переможе значення з другого словника.

Видалення елемента словника здійснюється за допомогою функції del():

>>> groups = {‘ГКЗ 2017-1’: ‘26’, ‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2016-1’: ‘19’}

>>> del(groups [‘ГКЗ 2017-1’])

>>> groups

{‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2016-1’: ‘19’}

Важливо: щоб видалити усі ключі та значення зі словника, слід скористатися функцією clear() або просто присвоїти порожній словник заданому імені.

Якщо необхідно дізнатися, чи міститься в словнику якийсь ключ, то застосовують ключове слово in:

>>> groups = {‘ГКЗ 2017-1’: ‘26’, ‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2016-1’: ‘19’}

>>> ‘ГКЗ 2017-1’ in groups

True

>>> ‘ГКЗ 2018-1’ in groups

False

Важливо: щоб уникнути помилок під час роботи із словником, спочатку необхідно перевірити чи є ключ у словнику або скористатися функцією get().

Щоб отримати усі ключі словника можна використати функцію keys():

```
>>> groups = {‘ГКЗ 2017-1’: ‘26’, ‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2016-1’: ‘19’}
>>> groups.keys()
dict_keys([‘ГКЗ 2016-1’, ‘ГКЗ 2017-1’, ‘ГКЗ 2017-1у’])
```

Важливо: у мові програмування Python 3 функція keys() повертає не список ключів, а dict_keys() – ітеративне подання ключів.

Ітеративне подання ключів зручно для великих словників, оскільки воно не вимагає часу та пам’яті для створення і збереження списку з цими ключами, але найчастіше необхідний саме список. У мові програмування Python 3 слід викликати функцію list(), щоб перетворити dict_keys у список.

Отримання усіх ключів словника у вигляді списку з використанням функції list() має вигляд:

```
>>> groups = {‘ГКЗ 2017-1’: ‘26’, ‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2016-1’: ‘19’}
>>> list(groups.keys())
[‘ГКЗ 2016-1’, ‘ГКЗ 2017-1’, ‘ГКЗ 2017-1у’]
```

У мові програмування Python 3 функція list() використовується для перетворення результатів роботи функцій values() та items() у звичайні списки.

Важливо: щоб отримати усі значення словника, необхідно використати функцію values():

```
>>> groups = {‘ГКЗ 2017-1’: ‘26’, ‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2016-1’: ‘19’}
>>> list(groups.values())
[‘9’, ‘26’, ‘19’]
```

Важливо: щоб отримати усі пари «ключ: значення» зі словника, необхідно використати функцію items():

```
>>> groups = {‘ГКЗ 2017-1’: ‘26’, ‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2016-1’: ‘19’}
>>> list(groups.items())
[(‘ГКЗ 2017-1у’, ‘9’), (‘ГКЗ 2017-1’, ‘26’), (‘ГКЗ 2016-1’, ‘19’)]
```

Отже, кожна пара «ключ: значення» повернулася у вигляді кортежу.

Як зазначалося раніше, у словнику абсолютно не важливий порядок пар, інтерпретатор виводить їх у випадковому порядку, тому постає питання, як отримати доступ до певного елемента, якщо індексація не можлива в принципі.

Для вирішення даної проблеми враховано, що у словнику доступ до значень здійснюється по ключам, які знаходяться у квадратних дужках (по аналогії з індексами рядків та списків):

```
>>> groups = {‘ГКЗ 2017-1’: ‘26’, ‘ГКЗ 2017-1у’: ‘9’, ‘ГКЗ 2016-1’: ‘19’}
>>> groups[‘ГКЗ 2017-1’]
‘26’
>>> groups[‘ГКЗ 2016-1’]
‘19’
```

Слід зазначити, що типи даних ключів та значень словників не обов’язково повинні бути рядками. Значення словників можуть бути більш складними (містити структури даних, наприклад, інші словники або списки).

Множини у мові програмування Python

Множина схожа на словник, у якого відсутні значення, тобто вона має тільки ключі. Як і у випадку зі словником, ключі повинні бути унікальними.

Щоб створити множину, необхідно використовувати функцію set() або розмістити у фігурних дужках одне або кілька значень, розділених комами:

```
>>> groups = set()
>>> groups
set()
>>> groups = {‘ГКЗ 2017-1’, ‘ГКЗ 2017-1у’, ‘ГКЗ 2016-1’}
>>> groups
{‘ГКЗ 2017-1у’, ‘ГКЗ 2017-1’, ‘ГКЗ 2016-1’}
```

Важливо: як і у випадку зі словником, порядок ключів у множині не має значення.

Крім того, можна створити множину за допомогою функції set() зі списку, рядка, кортежу або словника, втративши всі значення, що повторюються:

```
>>> set(‘hello’)
{‘l’, ‘e’, ‘h’, ‘o’ }
```

Важливо: множина у прикладі містить тільки одне входження літери l, незважаючи на те, що у слові hello є два входження цієї літери.

Складені структури даних у мові програмування Python

Різні типи даних (булеві значення, числа та рядки, списки, кортежі, множини та словники) можна об'єднувати у власні структури, більші та складніші.

Для прикладу, візьмемо три різні списки:

>>> groups = [‘ГКЗ 2017-1’, ‘ГКЗ 2017-1у’, ‘ГКЗ 2016-1’, ‘ГКЗ 2015-1’, ‘ГКЗ 2015-2’, ‘ГКЗ 2014-1’, ‘ГКЗ 2014-2’, ‘ГКЗ 2014-3’, ‘М ГКЗ 2017-1’]

>>> abetka = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]

>>> numbers = [‘one’, ‘two’, ‘three’]

Можна створити кортеж, що містить як елементи кожен з цих списків:

>>> cortege = groups, abetka, numbers

>>> cortege

([‘ГКЗ 2017-1’, ‘ГКЗ 2017-1у’, ‘ГКЗ 2016-1’, ‘ГКЗ 2015-1’, ‘ГКЗ 2015-2’, ‘ГКЗ 2014-1’, ‘ГКЗ 2014-2’, ‘ГКЗ 2014-3’, ‘М ГКЗ 2017-1’], [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’], [‘one’, ‘two’, ‘three’])

Можна створити список, що містить три списки:

>>> list = [groups, abetka, numbers]

>>> list

[[‘ГКЗ 2017-1’, ‘ГКЗ 2017-1у’, ‘ГКЗ 2016-1’, ‘ГКЗ 2015-1’, ‘ГКЗ 2015-2’, ‘ГКЗ 2014-1’, ‘ГКЗ 2014-2’, ‘ГКЗ 2014-3’, ‘М ГКЗ 2017-1’], [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’], [‘one’, ‘two’, ‘three’]]

Створимо словник зі списків:

>>> dictionary = {‘Pulpit’: groups, ‘Letters’: abetka, ‘Numbers’: numbers}

>>> dictionary

{‘Pulpit’: [‘ГКЗ 2017-1’, ‘ГКЗ 2017-1у’, ‘ГКЗ 2016-1’, ‘ГКЗ 2015-1’, ‘ГКЗ 2015-2’, ‘ГКЗ 2014-1’, ‘ГКЗ 2014-2’, ‘ГКЗ 2014-3’, ‘М ГКЗ 2017-1’], ‘Letters’: [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’], ‘Numbers’: [‘one’, ‘two’, ‘three’]}

Важливо: власні ключі словника повинні бути незмінними, тому список, словник чи множина не можуть бути ключем для іншого словника, однак, кортеж може бути ключем.

2 ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ АЛГОРИТМІВ НА МОВІ ПРОГРАМУВАННЯ PYTHON

2.1 Алгоритмічні структури у мові програмування Python

План

1. Структури коду в мові програмування Python.
2. Створення та перевірка умов у мові програмування Python.
3. Розгалуження у мові програмування Python.
4. Створення та перевірка повторень у мові програмування Python.
5. Включення (скорочення синтаксису) у мові програмування Python
6. Генератори у мові програмування Python.
7. Функції у мові програмування Python.
8. Виняткові ситуації під час обробки помилок.

Структури коду в мові програмування Python

Враховуючи, що програма – це множина інструкцій, то, відповідно до поставленої задачі, можна пропустити певні інструкції, повторити їх або вибрати одну із декількох для запуску. Отже, у програмі часто доводиться перевіряти умови та приймати рішення в залежності від цих умов.

Важливо: на блок-схемах, зазвичай, показують декілька можливих маршрутів від початку і до кінця перебігу якогось процесу або явища, ходу розв’язування задачі тощо.

Слід зазначити, що блок-схеми використовують і для побудови структури коду комп’ютерної програми. Кінцевий та початковий блоки програми подають округленими прямокутниками, блоки умов, в яких відбувається розгалуження програми, позначаються на блок-схемах ромбами, блоки з діями відображають прямокутниками.

Створення та перевірка умов у мові програмування Python

Для створення та перевірки умов у мові програмування Python використовують *булеві значення*, *оператори порівняння* та *булеві оператори*.

Як зазначалося раніше, булевий (логічний) тип даних може приймати лише два значення: True (істина) та False (хибне) [1].

Важливо: булевий тип даних отримав свою назву на честь англійського математика, засновника математичної логіки Джорджа Буля.

Оператори порівняння (табл. 2.1) порівнюють два значення між собою і повертають результат у вигляді булевого значення: істину або хибне [2].

Таблиця 2.1 – Оператори порівняння

Оператор	Назва
>	більше, ніж
<	менше, ніж
>=	більше або дорівнює
<=	менше або дорівнює
==	дорівнює
!=	не дорівнює

Присвоїмо змінній *a* цілочисельне значення 100:

```
>>> a = 100
>>> a == 55
False
>>> a == 100
True
>>> 25 < a
True
>>> a > 150
False
>>> ‘numbers’ == ‘numbers’
True
>>> ‘numbers’ == ‘Numbers’
False
>>> ‘numbers’ != ‘beasts’
True
>>> 20 == 20.0
True
>>> 20 == ‘20’
False
>>> 100 >= 20
True
```

Якщо потрібно виконати кілька порівнянь одночасно, то необхідно використати булеві (логічні) оператори *and*, *or* чи *not*, для визначення підсумкового результату.

Слід зазначити, що булеві оператори мають більш низький пріоритет, ніж фрагменти коду, які вони порівнюють, тобто спочатку вираховується результат фрагментів, а потім вони порівнюються.

Важливо: булеві оператори and та or завжди працюють з двома булевими значеннями (або виразами), тому їх називають бінарними.

Слід зазначити, що для роботи з булевими операторами *and* та *or* використовують таблиці істинності (табл. 2.2, табл. 2.3) [1].

Таблиця 2.2 – Таблиця істинності для оператора *and*

Вираз	Результат
True and True	True
True and False	False
False and True	False
False and False	False

Таблиця 2.3 – Таблиця істинності для оператора *or*

Вираз	Результат
True and True	True
True and False	True
False and True	True
False and False	False

Булевий оператор *not* змінює булеве значення на протилежне (табл. 2.4).

Таблиця 2.4 – Таблиця істинності для оператора *not*

Вираз	Результат
not True	False
not False	True

Важливо: булевий оператор *not* завжди діє на одне булеве значення або вираз, тому його називають *унарним*.

Приклади спрацювання булевих операторів *and*, *or* та *not*:

```
>>> True and True
True
>>> True and False
False
>>> True or False
True
>>> not True
False
```

Поєднання булевих значень, булевих операторів та порівнювання:

```
>>> (10 < 20) and (5 < 15)
True
>>> (5 < 15) and (10 < 5)
False
>>> (30 == 60) or (30 == 30)
True
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

Слід зазначити, що спочатку обчислюється лівий вираз, а потім – правий. Коли обидва результати стають відомими, обчислюється кінцевий результат усього виразу в цілому, який прийме вигляд єдиного булевого значення.

Крім того, як і для математичних операторів, для булевих операторів визначено порядок їх виконання. Після того, як будуть виконані усі математичні оператори та оператори порівнювання, першими виконуються оператори *not*, потім – оператори *and*, останніми – оператори *or*.

Слід зазначити, що будь-який булевий вираз може розглядатися як умова, обчислення якої завжди дає булеве значення True (істина) або False (хибне).

Розгалуження у мові програмування Python

У залежності, яке значення отримує умова, програма може виконувати або пропускати фрагменти коду – блоки коду, тобто має місце *розгалуження*.

Важливо: ознакою блоку коду є збільшення відступу від лівого краю. Блоки можуть містити інші блоки. Ознакою кінця блоку слугує зменшення відступу до нуля або до величини відступу зовнішнього блоку.

Отже, потрібно навчитися чітко виявляти істину або хибне, якщо елемент в умові, який перевіряється, не є булевим.

Наприклад, до False (хибне) прирівнюються такі значення [1]:

– булева змінна False;

– значення None;

– ціле число 0;

– число з плаваючою точкою 0.0;

– порожній рядок ‘’;

– порожній список [];

– порожній кортеж ();

– порожній словник {};

– порожня множина set().

Усі інші значення прирівнюються до True (істина).

Команда if у мові програмування Python

Завдяки команді if перевіряється поточний стан програми та вибираються подальші дії у залежності від результатів перевірки.

Синтаксис команди if такий:

```
if умова :
        блок коду
```

Роботу команди if можна описати таким чином: «Якщо умова істинна, то необхідно виконати даний блок коду».

Така форма розгалуження у програмуванні називається *неповною*.

Слід зазначити, що, якщо результат обчислення умови є істинним (True), то блок коду виконується, якщо результат обчислення умови є хибним (False), то блок коду не буде виконуватися.

У мові програмування Python інструкція if включає такі елементи:

– ключове слово if;

– умова (вираз, обчислення якого дає одне з двох значень: True або False);

– двокрапка;

– блок коду з відступом, який починається в наступному рядку.

Припустимо, є програмний код, що перевіряє правильність введення певного прізвища, наприклад, Ivanov:

```
if surname == ‘Ivanov’ :
        print(‘Hello, Ivanov.’)
```

Якщо вважати, що змінній surname попередньо присвоєно значення Ivanov, то після виконання фрагмента коду, отримаємо на екрані повідомлення:

```
print(‘Hello, Ivanov.’)
```

У протилежному випадку, якщо змінна surname матиме інше значення, блок коду команди if, що складається з однієї інструкції print(‘Hello, Ivanov.’), не буде виконано і жодного повідомлення на екрані не з’явиться.

Блок-схема даного фрагмента коду подана на рисунку 2.1 [1].

Команда else у мові програмування Python

За блоком коду команди if може слідувати необов’язкова команда else із власним блоком коду, який виконується лише у тому випадку, якщо умова if є хибною.

Важливо: така форма розгалуження у програмуванні називається *повною*.

if умова :

блок коду, коли умова істинна

else:

блок коду, коли умова хибна

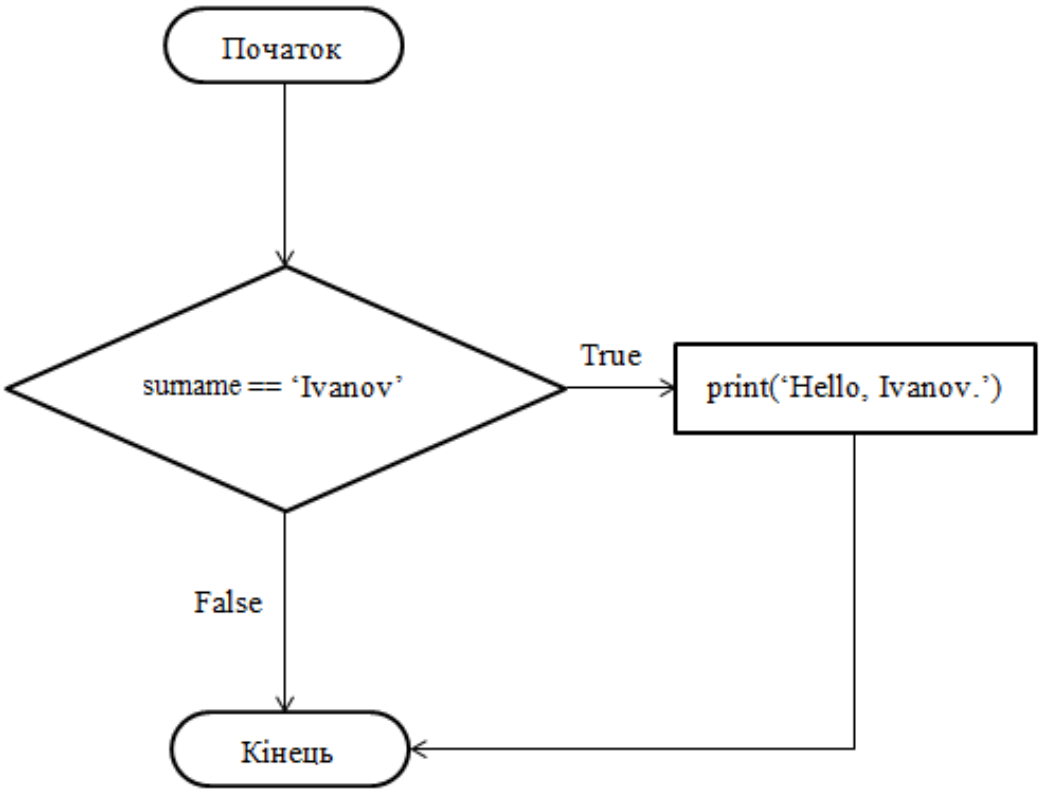

Рисунок 2.1 – Блок-схема неповної форми розгалуження: інструкція if

Конструкцію if/else слід розуміти так: «Якщо умова істинна, то виконати даний блок коду, у протилежному випадку виконати наступний блок коду».

У мові програмування Python команда else не має умови та складається із таких елементів:

– ключове слово else;

– двокрапка;

– блок коду з відступом, що починається на наступному рядку.

Розглянемо код, що має інструкцію else, яка виводить інше повідомлення, якщо змінній surname попередньо надано значення не Ivanov, а, наприклад, Petrov:

```
if surname == 'Ivanov' :
    print('Hello, Ivanov.')
else:
    print('Hello, user.')
```

Результатом виконання даного фрагмента коду буде повідомлення із блоку коду команди else, що складається із однієї інструкції print(‘Hello, user.’):

Hello, user.

Блок-схема даного фрагмента коду подана на рисунку 2.2 [1].

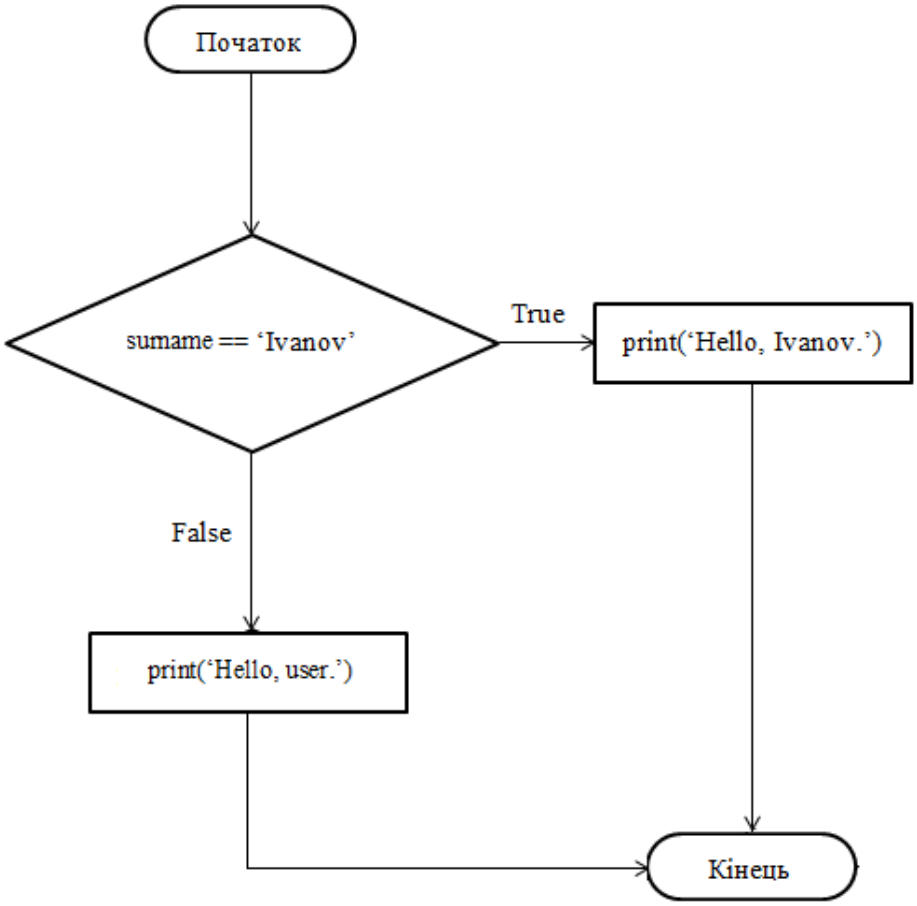

Рисунок 2.2 – Блок-схема повної форми розгалуження: конструкція if/else

Команда elif у мові програмування Python

Для перевірки більше однієї умови застосовують команду elif (скорочено від else if). Записується дана команда тільки після інструкцій if або після іншої команди elif, вона дозволяє вказувати додаткові умови для перевірки [2]:

```
if умова1 :
        блок коду, коли умова1 істинна
elif умова2:
        блок коду, коли умова2 істинна
elif умова3:
        блок коду, коли умова3 істинна
і т. д.
```

У мові програмування Python команда elif завжди складається з таких елементів:

– ключове слово elif;

– умова (вираз, обчислення якого дає одне з двох значень: True або False);

– двокрапка;

– блок коду з відступом, що починається на наступному рядку.

Важливо: у випадку ланцюжка команд elif, як тільки з'ясовується, що одна з умов істинна, всі інші блоки elif автоматично пропускаються, тобто буде виконано або тільки один блок коду, або жоден з них.

Команда if/elif/else у мові програмування Python

При необхідності за останньою інструкцією elif розміщують команду else, у даному випадку гарантується, що буде виконано хоча б один (і тільки один) блок коду:

```
if умова1 :
        блок коду, коли умова1 істинна
elif умова2:
        блок коду, коли умова2 істинна
elif умова3:
        блок коду, коли умова3 істинна
else:
        блок коду, коли усі умови хибні
```

Слід зазначити, що, якщо умови усіх інструкцій if та elif будуть хибними, то виконається блок коду інструкції else.

Конструкцію if/elif/else слід розуміти так: «Якщо перша умова істинна, то виконати даний блок коду. Якщо друга умова істинна, виконати даний блок коду. У протилежному випадку виконати даний блок коду».

Створення та перевірка повторень у мові програмування Python

Перевірки за допомогою if, elif та else виконуються послідовно, інколи необхідно виконати певні операції більше одного разу, тому створюють *цикл*.

Команда while у мові програмування Python

Одним із варіантів створення циклів у мові програмування Python є while:

```
while умова :
        блок коду, коли умова істинна
```

У мові програмування Python команда while завжди складається з таких елементів:

– ключове слово while;

– умова (вираз, обчислення якого дає одне з двох значень: True або False);

– двокрапка;

– блок коду з відступом, що починається на наступному рядку.

Команда while має схожу структуру із командою if, але веде себе по іншому: під час досягнення кінця блоку коду команди if, керування виконанням програми передається наступній команді, а під час досягнення кінця блоку коду команди while, керування передається на початок циклу і програма продовжує знову виконувати той самий блок коду.

Важливо: у циклі while умова завжди перевіряється на початку кожної ітерації (ітерація – крок виконання циклу):

```
number = 1
while number <= 10:
        print(number)
        number += 1
```

Даний цикл виводить на екран числа від 1 до 10:

```
1
2
3
4
5
6
7
8
9
10
```

У продемонстрованому прикладі на першому етапі присвоюється значення 1 змінній number. Цикл while порівнює значення змінної number із числом 10 та продовжує роботу, до тих пір, поки значення number є меншим або дорівнює 10 (умова циклу є істинною), у протилежному випадку цикл завершується (умова циклу є хибною).

Усередині циклу (у тілі циклу) виводиться значення змінної number. Далі збільшується значення змінної number на 1 за допомогою виразу number += 1.

Мова програмування Python повертається до початку циклу і знову порівнює значення змінної number з числом 10. Значення змінної number на другому етапі дорівнює 2, тому тіло циклу while виконується знову і змінна number збільшується до 3. Виконання продовжується до тих пір, поки значення змінної number не буде дорівнювати 11 у тілі циклу.

Під час чергового повернення на початок циклу перевірка number <= 10 поверне значення False і цикл while закінчується. Мова програмування Python перейде до виконання наступних рядків коду.

Переривання циклу (break) у мові програмування Python

Якщо необхідно, щоб цикл виконувався до тих пір, поки щось не станеться, але точно невідомо, коли ця подія трапиться, можна скористатися нескінченним циклом, що містить оператор break. Якщо програма у процесі виконання досягає команди break, то спрацювання циклу відразу припиняється.

Слід зазначити, що рядок while True створює нескінченний цикл – умова такого циклу завжди істинна. Програма буде виконувати команди циклу та вийде з нього тільки у тому випадку, якщо виконається інструкція break, помилково думати, що з нескінченного циклу вийти неможливо.

Нескінченний цикл та вихід з нього у мові програмування Python

Якщо програма потрапила в нескінченний цикл, то необхідно натиснути комбінацію клавіш Ctrl+C, і вона припинить своє виконання.

Перевіримо, як працює нескінченний цикл:

```
while True:
    print('Hello, student!')
```

Дана програма е процесі виконання без зупинки буде повторювати виведення повідомлення Hello, student!, оскільки умова циклу while завжди істинна (True).

Щоб попередити зациклювання програми, необхідно, щоб умова циклу набула значення False або виконувалася команда break, наприклад, якщо у коді:

```
number = 1
while number <= 10:
    print(number)
    number += 1
```

Якщо пропустити рядок number += 1, то цикл стане нескінченним:

```
number = 1
while number <= 10:
    print(number)
1
1
1
...
```

Продовження циклу (continue) у мові програмування Python

Іноді потрібно не переривати весь цикл, а лише пропустити, по певній причині, одну ітерацію. Для таких ситуацій використовується оператор continue.

Коли програма у процесі виконання досягає інструкції continue, відразу ж керування програмою передається на початок циклу, де умова перевіряється знову (дана операція відбувається і при звичайному досягненні кінця циклу).

Цикл for та функція range() у мові програмування Python

Цикл while продовжує виконуватися доти, доки умова залишається істинною, якщо ж необхідно виконати блок коду відому кількість разів, то використовують цикл for та функцію range():

```
for змінна in range() :
        блок коду
```

У мові програмування Python команда for складається з таких елементів:

– ключове слово for;

– ім’я змінної;

– ключове слово in;

– виклик функції range(), в яку можна передати до трьох цілих чисел, розділених комами;

– двокрапка;

– блок коду з відступом, що починається на наступному рядку.

Приклад спрацювання циклу for:

```
print(‘My name is’)
for i in range(10):
        print(‘Iryna Ten Times (‘ + str(i) + ’)’)
```

Блок коду циклу for виконується 10 разів. На першій ітерації (під час першого виконанні) значення змінної встановлюється рівним 0.

Виклик функцій print() у тілі циклу виводить повідомлення Iryna Ten Times (0).

Коли цикл закінчує ітерацію, виконавши увесь блок коду, управління передається на початок циклу, де інструкція for збільшує значення змінної на 1.

Виклик функції range(10) забезпечує десятикратне виконання блоку коду циклу, встановлюючи для змінної послідовно значення 0, 1, 2, 3, 4, 5, 6, 7, 8 і 9.

Важливо: значення 10, вказане у дужках функції range(), до послідовності значень змінної не входить.

Коли запустити дану програму, вона виведе десять рядків тексту, кожен буде закінчуватись поточним значенням змінної, і цикл закінчить свою роботу:

```
My name is
Iryna Ten Times (0)
Iryna Ten Times (1)
Iryna Ten Times (2)
Iryna Ten Times (3)
Iryna Ten Times (4)
Iryna Ten Times (5)
Iryna Ten Times (6)
Iryna Ten Times (7)
Iryna Ten Times (8)
Iryna Ten Times (9)
```

Крім того, все те, що робить цикл for, можна зробити і за допомогою циклу while:

```
print('My name is')
i = 0
while i < 10:
    print('Iryna Ten Times (' + str(i) + ')')
    i = i + 1
```

Слід зазначити, що у функцію range() можна передавати аргументи – значення трьох цілих чисел, розділених комами.

Перше число вказує, з якого значення починає змінюватися змінна циклу for, а друге число – значення, до якого може змінюватися змінна циклу for (число не входить у діапазон):

```
for i in range(12, 16):
    print(i)
12
13
14
15
```

Функцію range() можна викликати і з трьома аргументами.

Перші два – вказують початок та кінець діапазону зміни значень змінної циклу for, а третій задає крок цієї зміни.

```
for i in range(1, 10, 2):
    print(i)
1
3
5
7
9
```

Функцію range() можна викликати, задавши від'ємний крок.

У такому випадку значення змінної циклу for буде змінюватися від більших значень до менших:

```
for i in range(3, -4, -1):
    print(i)
3
2
1
0
-1
-2
-3
```

Слід зазначити, що цикл for дозволяє проходити по різним структурам даних, які є послідовностями. Наприклад, використання циклу for для списку:

```
birds = ['pigeon', 'crow', 'owl', 'eagle']
for bird in birds:
    print(bird)
pigeon
crow
owl
eagle
```

Списки є прикладом ітеративних об'єктів у мові програмування Python поряд з рядками, кортежами, словниками.

Важливо: ітерація по кортежу, списку або рядку повертає один елемент за один раз:

```
word = 'cadastre'
for letter in word:
    print(letter)
```

c
a
d
a
s
t
r
e

Важливо: ітерація по словнику у циклі (або використовуючи функцію keys()) повертає ключі словника:

```
professions = {‘it’: ‘web developer’, ‘education’: ‘teacher’}
for key in professions: # або for key in professions.keys():
    print(key)
it
education
```

Важливо: щоб виконувати ітерацію по значенням словника, а не по ключам, необхідно використовувати функцію values():

```
professions = {‘it’: ‘web developer’, ‘education’: ‘teacher’}
for value in professions.values():
    print(value)
web developer
teacher
```

Важливо: для отримання ключа та значення із словника, можна використовувати функцію items(), результатом буде кортеж із парами «ключ: значення»:

```
professions = {‘it’: ‘web developer’, ‘education’: ‘teacher’}
for item in professions.items():
    print(item)
(‘it’, ‘web developer’)
(‘education’, ‘teacher’)
```

Слід зазначити, що можна присвоїти значення кортежу за один крок.

Для кожного кортежу, який повертає функція items(), необхідно надати перше значення (ключ) змінній key, а друге (значення) – змінній value.

```
professions = {‘it’: ‘web developer’, ‘education’: ‘teacher’}
for key, value in professions.items():
    print(‘Category’, key, ‘has a profession’, value)
Category it has a profession web developer
Category education has a profession teacher
```

Словник завжди підтримує зв’язок між ключем та пов’язаним з ним значенням, але порядок отримання елементів із словника непередбачуваний.

Слід зазначити, що один із способів отримання елементів у певному порядку заснований на сортуванні ключів, що повертаються циклом for.

Для отримання упорядкованої копії ключів можна скористатися функцією sorted():

```
favorite_languages = {‘Iryna’: ‘Python’, ‘Olga’: ‘C’, ‘Igor’: ‘Python’}
for name in sorted(favorite_languages.keys()):
    print(name.title() + ‘‘, thank you for taking the poll.’’)
```

Така конструкція видає список усіх ключів словника, але упорядковує їх перед тим, як перебрати елементи. У результаті на екрані відображаються перераховані усі користувачі, що брали участь в опитуванні, щодо улюблених мов програмування, а їх імена упорядковані за алфавітом:

```
Igor, thank you for taking the poll.
Iryna, thank you for taking the poll.
Olga, thank you for taking the poll.
```

Функція zip() у мові програмування Python

Важливо: функція zip() використовується для паралельної ітерації по декільком послідовностям одночасно.

Приклад застосування функції zip():

```
days = [‘Monday’, ‘Tuesday’, ‘Wednesday’, ‘Thursday’]
fruits = [‘coconut’, ‘lemon’, ‘mango’, ‘apple’]
drinks = [‘coffee’, ‘tea’, ‘fruit juice’, ‘milk’]
desserts = [‘marmalade’, ‘ice cream’, ‘pie’, ‘pudding’, ‘candy’]
for day, fruit, drink, dessert in zip(days, fruits, drinks, desserts):
    print(day, “: drink”, drink, “eat”, fruit, “enjoy”, dessert)
Monday : drink coffee eat coconut enjoy marmalade
Tuesday : drink tea eat lemon enjoy ice cream
Wednesday : drink fruit juice eat mango enjoy pie
Thursday : drink milk eat apple enjoy pudding
```

Слід зазначити, що функція zip() припиняє свою роботу у випадку використання найкоротшої послідовності (у наведеному прикладі один із списків (desserts) виявився довшим за інші, тому ніхто не отримає candy, поки не збільшаться інші списки).

Важливо: функцію zip() можна використати, щоб пройти по декільком послідовностям та створити кортежі елементів із однаковими індексами.

Для прикладу створено два кортежі відповідних один одному англійських та німецьких слів:

```
english = ‘Monday’, ‘Tuesday’, ‘Wednesday’
german = ‘Montag’, ‘Dienstag’, ‘Mittwoch’
```

Використаємо функцію zip(), щоб об’єднати дані кортежі в пару.

Слід зазначити, що значення, яке повертається функцією zip(), саме по собі не є списком або кортежем, але його можна перетворити у будь-яку з цих послідовностей, наприклад, у список, з використанням функції list():

```
print(list(zip(english, german)))
[(‘Monday’, ‘Montag’), (‘Tuesday’, ‘Dienstag’), (‘Wednesday’, ‘Mittwoch’)]
```

Крім того, якщо передати результат роботи функції zip() безпосередньо функції dict(), то отримаємо готовий невеликий англо-німецький словник:

```
print(dict(zip(english, german)))
{‘Wednesday’: ‘Mittwoch’, ‘Monday’: ‘Montag’, ‘Tuesday’: ‘Dienstag’}
```

Включення (скорочення синтаксису) у мові програмування Python

Включення дозволяють об’єднувати цикли та умовні перевірки, не використовуючи при цьому громіздкий синтаксис, це одна із особливостей мови програмування Python.

Включення для списків у мові програмування Python

Приклад створення числового списку:

```
number_list = []
for number in range(1, 6):
    number_list.append(number)
print(number_list)
[1, 2, 3, 4, 5]
```

Форма застосування включення для створення числового списку:

[вираз for елемент in ітеративний елемент]

Приклад створення списку цілих чисел через включення списку:

```
number_list = [number for number in range(1,6)]
print(number_list)
[1, 2, 3, 4, 5]
```

У даному прикладі включення списку переміщує цикл у квадратні дужки.

Перша змінна number (вона є виразом) формує значення для списку number_list. Друга змінна number є частиною циклу for. Щоб показати, що перша змінна number є виразом, змінимо задачу таким чином:

```
number_list = [number-1 for number in range(1,6)]
print(number_list)
```

Отримаємо список із значеннями меншими на 1:

```
[0, 1, 2, 3, 4]
```

Включення списку може містити умовний вираз:

```
[вираз for елемент in ітеративний елемент if умова]
```

Переглянемо нове включення, яке створює список, що складається тільки з непарних чисел, розташованих у діапазоні від 1 до 5.

Важливо: використовуйте умови number % 2 == 1 або number % 2 == 0 для відбору відповідно непарних або парних чисел:

```
a_list = [number for number in range(1, 6) if number % 2 == 1]
print(a_list)
```

Отримуємо список непарних чисел:

```
[1, 3, 5]
```

Включення списку з умовою є більш компактним, ніж звичайний код:

```
a_list = []
for number in range(1, 6):
    if number % 2 == 1:
        a_list.append(number)
print(a_list)
```

Отримуємо список непарних чисел:

```
[1, 3, 5]
```

Включення для словників у мові програмування Python

Для словників створення включення виглядає так:

{ключ: значення for вираз in ітеративний елемент}

До включення словників також входять перевірки if та оператори for:

```
word = ‘Programming’
letter_counts = {letter: word.count(letter) for letter in word}
print(letter_counts)
{‘P’: 1, ‘r’: 2, ‘o’: 1, ‘g’: 2, ‘a’: 1, ‘m’: 2, ‘i’: 1, ‘n’: 1}
```

У даному циклі аналізується слово Programming та обчислюється скільки разів з’являється кожна поточна буква. Команда word.count(letter) викликається декілька разів, так як букви r, g та m зустрічаються по два рази.

Однак, на вміст словника це не впливає, бо коли проходимо букви p, o, a, i або n другий раз, то існуючий запис у словнику замінюється на поточне значення.

Розв’язання попередньої задачі традиційним способом буде таким:

```
word = ‘Programming’
letter_counts = {}
for letter in set(word):
    letter_counts[letter] = word.count(letter)
print(letter_counts)
{‘a’: 1, ‘m’: 2, ‘i’: 1, ‘n’: 1, ‘P’: 1, ‘r’: 2, ‘o’: 1, ‘g’: 2}
```

Генератори у мові програмування Python

У мові програмування Python *генератор* – це об’єкт, який призначений для створення послідовностей [1].

За допомогою генераторів можна досліджувати великі послідовності без створення та збереження у пам’яті усієї послідовності відразу.

Генератори є джерелом даних для ітераторів.

Використаємо генератор – функцію range(), що має послідовність цілих чисел та обчислює суму чисел від 1 до 50 включно за допомогою функції sum():

```
>>> sum(range(1,51))
1275
```

Функції у мові програмування Python

Функції – це іменовані блоки коду, призначені для вирішення однієї конкретної задачі [6].

Якщо певна задача повинна розв'язуватися багаторазово у програмі, то не потрібно декілька разів вводити необхідний код, для цього викликається функція, що призначена для розв'язання даної задачі, цей виклик вимагає від мови програмування Python виконати код, що міститься усередині функції.

Важливо: функція – це міні-програма у межах основної програми.

Визначення і виклик функцій у мові програмування Python

Мова програмування Python використовує вбудовані функції, наприклад, len(), print(), input() тощо, однак, можна створювати і власні функції.

Використання функцій спрощує читання, написання, тестування коду, а також виправлення помилок.

Створення функцій – це перший крок до повторного використання коду.

Крім того, функція може приймати будь-яку кількість будь-яких вхідних параметрів та повертати будь-яку кількість будь-яких результатів.

Слід зазначити, що функцію можна [9]:

– визначити;

– викликати.

Для визначення функції, необхідно написати def, ім'я функції, вхідні параметри у круглих дужках, двокрапку, після якої з відступом йде блок коду:

```
def назва_функції(вхідні параметри):
    блок коду
```

Важливо: під час написання імен функцій дотримуються аналогічних правил, що і для імен змінних: імена повинні починатися з букви або знака підкреслення (містити тільки букви, цифри або знак підкреслення).

Для прикладу визначимо функцію без вхідних параметрів, яка виводить повідомлення на екран:

```
def question_friend():
    print('How are you?')
```

Для виклику функції необхідно написати її ім'я, круглі дужки, наприклад, question_friend(). Коли викликається функція question_friend(), Python виконує код усередині функції – виводить питання: How are you? на екран та повертає назад керування основній програмі.

Розглянемо функцію без параметрів, яка повертає значення, та викличемо цю функцію в команді if для перевірки значення, яке повертає функція:

```
def frozen():
    return True
if frozen():
    print('Wear warm clothes!')
else:
    print('Jump on the spot.')
```

Комбінація функції frozen() з перевіркою відповідної умови виведе повідомлення: «Wear warm clothes!».

Визначимо функцію travel(), яка має один параметр country. Дана функція використовує оператор return, щоб відправити результат виконання у зовнішню основну програму (звідки викликається функція):

```
def travel(country):
    return country
```

Викличемо функцію travel(), передавши їй рядок: 'I enjoy travelling!'.

Функція print() використовується для виведення на екран значення, яке повернула функція оператором return:

```
print(travel('I enjoy travelling!'))
```

Отримаємо такий результат: I enjoy travelling!

Слід зазначити, що значення, які передаються у функцію під час виклику, називаються *аргументами*, коли викликається функція з аргументами, значення цих аргументів копіюються у відповідні параметри всередині функції [10].

Отже, у даному прикладі зазначеній функції travel() передавалось одне значення: рядок 'I enjoy travelling!'. Це значення копіювалось усередині функції travel() у параметр country, а потім поверталось у зовнішню програму, звідки викликалась функція.

Значення None у мові програмування Python

Для прикладу визначимо функцію, яка не має параметрів:

```
def do_nothing():
    pass
```

Важливо: вираз pass вказує мові програмування Python, що функція нічого не робить.

Викличемо зазначену функцію таким чином:

```
do_nothing()
```

Слід зазначити, що функція do_nothing() відпрацює, але нічого не виведе на екран.

Для отримання значення None викличемо дану функцію з використанням функції print():

```
print(do_nothing())
None
```

Важливо: якщо функція не використовує оператор return явно, то вона повертає результат None.

None – це спеціальне значення в мові програмування Python, яке означає нічого або порожнє місце, коли функція нічого не повертає [1].

Важливо: значення None не є булевим значенням False.

Для прикладу визначимо функцію, що виводить перевірку значення None:

```
def is_none(thing):
    if thing is None:
        print('It's None')
    elif thing:
        print('It's True')
    else:
        print('It's False')
```

Виконаємо декілька перевірок:

```
is_none(None)      # It's None
is_none(True)      # It's True
is_none(False)     # It's False
is_none(0)         # It's False
is_none(0.0)       # It's False
is_none(())        # It's False
is_none([])        # It's False
is_none({})        # It's False
is_none(set())     # It's False
```

Позиційні та іменовані аргументи у мові програмування Python

Слід зазначити, що під час визначення функції можна вказати декілька параметрів, а під час виклику функції можна передати їй декілька аргументів. Крім того, існує декілька способів передачі аргументів функціям.

Найбільш поширений тип аргументів – це *позиційні аргументи*, значення яких копіюються у відповідні параметри функції відповідно до порядку слідування, необхідно пам'ятати кожну позицію аргументів.

Визначимо функцію, яка створює та повертає словник із вхідних позиційних аргументів:

```
def thematic_layer(name, kind, number_spatial_objects):
    return{'name': name, 'kind': kind, 'number_spatial_objects': number_spatial_objects}
```

У даному прикладі під час визначення функції thematic_layer, передбачено її параметри у круглих дужках name, kind, number spatial objects.

Якщо викликати дану функцію командою print(thematic_layer('roads', 'linear', '280')), то саме значення roads, linear та 280 і є позиційними аргументами, які будуть скопійовані у відповідні параметри функції і вона поверне словник (слід пам'ятати, що порядок у словнику непередбачуваний):

```
{'number_spatial_objects': '280', 'kind': 'linear', 'name': 'roads'}
```

Аргументи у виклику функції можна вказувати також і за допомогою *іменованих аргументів* – імен відповідних параметрів. Порядок слідування аргументів, у цьому випадку, може бути яким завгодно:

```
def thematic_layer(name, kind, number_spatial_objects):
    return{'name': name, 'kind': kind, 'number_spatial_objects': number_spatial_objects}
print(thematic_layer(name = 'roads', kind = 'linear', number_spatial_objects = '280'))
```

Слід зазначити, що у даному виклику функції thematic_layer:

– name – це ім'я аргументу;

– 'roads' – значення аргументу.

У результаті словник набуде такого вигляду:

```
{'kind': 'linear', 'number_spatial_objects': '280', 'name': 'roads'}
```

Крім того, можна об’єднувати позиційні та іменовані аргументи, однак, позиційні аргументи завжди вказуються першими:

```
def thematic_layer(name, kind, number_spatial_objects):
    return{‘name’: name, ‘kind’: kind, ‘number_spatial_objects’: number_spatial_objects}
print(thematic_layer(‘roads’, kind = ‘linear’, number_spatial_objects = ‘280’))
```

Результат виведення набуде такого вигляду:

```
{‘kind’: ‘linear’, ‘name’: ‘roads’, ‘number_spatial_objects’: ‘280’}
```

Значення за замовчуванням у мові програмування Python

Для кожного параметра функції можна визначити значення за замовчуванням. Якщо під час виклику функції передається аргумент, який відповідає цьому параметру, то мова програмування Python застосовує значення аргументу, якщо його немає – застосовує значення за замовчуванням.

Таким чином, якщо для параметра визначено значення за замовчуванням, то можна вилучити відповідний аргумент, який, зазвичай, входить у виклик функції.

Визначимо функцію, що повертає інформацію про домашніх тварин:

```
def describe_pet(pet_name, animal_type=‘cat’):
    print(‘I have a ’ + animal_type + ‘.’)
    print(‘My ’ + animal_type + ‘’s name is ’ + pet_name.title() + ‘.’)
```

У визначення функції describe_pet() включений параметр animal_type із значенням за замовчуванням ‘cat’. Якщо функція буде викликана із іменованим аргументом pet_name=‘Liza’, але без аргумента animal_type:

```
describe_pet(pet_name=‘Liza’)
```

Мова програмування Python знає, що для параметра animal_type слід використовувати значення ‘cat’:

```
I have a cat.
My cat’s name is Liza.
```

Для виведення інформації про будь-яку іншу домашню тварину, окрім кішки, використовується виклик функції describe_pet() такого вигляду:

```
describe_pet(pet_name=‘Homka’, animal_type=‘hamster’)
```

Так як аргумент для параметра animal_type у виклику функції заданий явно (‘Homka’), мова програмування Python ігнорує значення параметра за замовчуванням:

I have a hamster.
My hamster’s name is Homka.

Особливості використання аргументів із символами «» та «**» у мові програмування Python*

Якщо символ «*» буде використано усередині функції з параметром, то довільну кількість позиційних аргументів буде згруповано у *кортеж*.

Приклад визначення функції print_days():

```
def print_days(*args):
    print(‘Get ready:’, args)
```

У даному прикладі args є кортежем параметрів.

Усі аргументи, які передаються у функцію print_days() будуть виведені на екран, як кортеж args:

```
print_days(‘Wednesday’, ‘Thursday’, ‘Friday...’)
Get ready: (‘Wednesday’, ‘Thursday’, ‘Friday...’)
```

Зазначений спосіб корисний під час написання функцій, які приймають довільну кількість аргументів, наприклад, print()

```
def print_travel(required1, required2, *args):
    print(‘For train travel is required:’, required1)
    print(‘For train travel is required too:’, required2)
    print(‘All the rest:’, args)
```

У даному прикладі обов’язковими параметрами є required1 та required2, до них будуть скопійовані значення позиційних аргументів (‘return ticket’ та ‘train’) під час виклику функції:

```
print_travel(‘return ticket’, ‘train’, ‘carriage’, ‘seat’, ‘luggage rack’)
```

Кортеж *args отримає усі інші аргументи, починаючи із значення ‘carriage’:

```
For train travel is required: return ticket
For train travel is required too: train
All the rest: (‘carriage’, ‘seat’, ‘luggage rack’)
```

Слід зазначити, що можна використовувати два символи **, щоб згрупувати іменовані аргументи у словник, у якому імена аргументів стануть ключами, а їх значення – відповідними значеннями у словнику:

```
def print_character(**kwargs):
    print('Person\'s characteristics:', kwargs)
print_character(emotions='cheerful', sense='sad', look='slim')
```

Виводяться її іменовані аргументи у вигляді словника:

```
Person's characteristics: {'emotions': 'cheerful', 'look': 'slim', 'sense': 'sad'}
```

Важливо: під час використання символу «*» не обов'язково називати кортеж параметрів args, а під час використання символів «**» називати словник kwargs.

Простір імен та області видимості у мові програмування Python

Програми на мові програмування Python можуть мати різні *простори імен* – розділи, усередині яких певне ім'я унікальне та не пов'язане з такими ж іменами в інших просторах імен.

Слід зазначити, що кожна функція визначає власний простір імен.

Якщо визначити змінну, яка називається *a* в основній програмі, та іншу змінну *a* в окремій функції, то вони будуть посилатися на різні значення.

Важливо: в основній програмі визначається глобальний простір імен, тому змінні, що знаходяться у цьому просторі імен, є *глобальними*.

Значення глобальної змінної можна отримати усередині функції:

```
thematic_layer = 'quarters'
def print_global():
    print('inside print_global:', thematic_layer)
```

Використавши функцію print() і виклик функції print_global(), отримаємо:

```
print('at the top level:', thematic_layer)
print_global()
```

Отримаємо значення змінної thematic_layer для обох випадків:

```
at the top level: quarters
inside print_global: quarters
```

Розглянемо програму з іншою функцією, яка містить змінну з такою ж назвою, що і глобальна змінна thematic_layer:

```
thematic_layer = ‘quarters’
def change_local():
    thematic_layer = ‘roads’
    print(‘inside change_local:’, thematic_layer, id(thematic_layer))

change_local()
print(thematic_layer)
print(id(thematic_layer))
```

У результаті виконання програми отримаємо:

```
inside change_local: roads 40
quarters
20
```

У наведеному прикладі присвоєно рядок ‘quarters’ глобальній змінній з іменем thematic_layer. Функція change_local() також має змінну thematic_layer, але ця змінна знаходиться у локальному просторі імен функції.

У четвертому ряді коду використано функцію id(), щоб вивести на екран унікальне значення об’єкта thematic_layer усередині функції, а у сьомому – щоб вивести на екран унікальне значення об’єкта thematic_layer ззовні функції.

Порівнюючи результати роботи функції id(), можна зробити висновок, що змінна thematic_layer, яка розміщена усередині функції change_local() – це не змінна, що знаходиться на рівні основної програми.

Щоб змінна усередині функції була видимою в основній програмі, необхідно явно використовувати ключове слово global:

```
thematic_layer = ‘quarters’
def change_and_print_global():
    global thematic_layer
    thematic_layer = ‘roads’
    print(‘inside change_and_print_global:’, thematic_layer)

print(thematic_layer)
change_and_print_global()
print(thematic_layer)
```

Отримаємо такі результати:

```
quarters
inside change_and_print_global: roads
roads
```

У наведеному прикладі до виклику функції change_and_print_global() глобальна змінна thematic_layer мала значення ‘quarters’.

Виклик функції change_and_print_global() з використанням ключового слова global у функції перетворив локальну змінну thematic_layer на глобальну та переписав її значення з ‘quarters’ на ‘roads’.

Змінна thematic_layer в основній програмі тепер має значення ‘roads’.

Важливо: якщо не використовувати ключове слово global усередині функції, то мова програмування Python використовує локальний простір імен, тому змінна буде локальною.

Слід зазначити, що змінна зникає після того, як функція завершує роботу.

Отже, мова програмування Python надає дві функції для доступу до вмісту просторів імен:

– locals() – повертає словник, що містить імена локального простору;

– globals() – повертає словник, що містить імена глобального простору.

Схема використання даних функцій така:

```
thematic_layer = ‘quarters’ (глобальна змінна)
def change_local():
    thematic_layer = ‘roads’ (локальна змінна)
    print(‘locals:’, locals())

print(thematic_layer)
change_local()
print(‘globals:’, globals())
print(thematic_layer)
```

Отримаємо такі результати:

```
quarters
locals: {‘thematic_layer’: ‘roads’}
globals: {‘thematic_layer’: ‘quarters’
quarters
```

У наведеному прикладі локальний простір імен усередині функції change_local() містить лише локальну змінну thematic_layer зі значенням ‘roads’.

Глобальний простір імен містить окрему змінну thematic_layer зі значенням ‘quarters’.

Виняткові ситуації під час обробки помилок

Для управління помилками, що виникають у ході виконання програми, у мові програмування Python використовуються спеціальні об’єкти, що називаються *винятками* [1].

Якщо під час виникнення помилки мова програмування Python не знає, що робити далі, створюється *об’єкт винятку*. Якщо у програму включено код обробки виняткової ситуації, то виконання програми продовжиться, а якщо ні, то програма зупиняється і виводить *трасування* (інформацію про хід виконання програми) зі звітом про помилку.

Даний факт можна спостерігати, коли спробувати отримати доступ до елемента, який не входить у список або кортеж, або отримати значення елемента у словнику по ключу, якого не існує.

Слід зазначити, що коли виконується код, який при деяких обставинах може не спрацювати, використовують *обробники винятків,* щоб перехопити будь-які потенційні помилки.

Якщо не використовувати обробники винятків, то мова програмування Python виведе повідомлення про помилку та деяку інформацію про те, де сталася помилка, а потім завершить програму.

Важливо: переглянути список винятків, які генеруються у разі появи помилок, можна на сайті офіційної документації мови програмування Python.

Можна також визначити власні типи винятків, щоб обробляти особливі ситуації, які можуть виникнути у програмах.

Важливо: для визначення власних винятків необхідно використовувати класи, які будуть детально розглянуті у підрозділі 3.1.

2.2 Модулі та пакети у мові програмування Python

План

1. Імпорт модулів: інструкція import у мові програмування Python.
2. Пакети у мові програмування Python.
3. Стандартна бібліотека мови програмування Python.

Будь-якій програмі на мові програмування Python доступний базовий набір вбудованих функцій, в число яких входять, наприклад, такі функції як print(), input(), len() тощо.

Крім того, у мову програмування Python входить набір модулів, який називається *стандартною бібліотекою* [2].

Наприклад, модуль math містить математичні функції, модуль random – функції для роботи з випадковими числами і т. д.

Імпорт модулів: інструкція import у мові програмування Python

Важливо: щоб використовувати функції, які входять у модуль, необхідно його імпортувати (підключити) у програму за допомогою інструкції import.

Схема використання інструкції import така:

```
import модуль
```

Слід зазначити, що у даній схемі модуль – це ім’я іншого файлу на мові програмування Python без розширення .py. Як тільки модуль буде імпортовано, можна використовувати будь-яку функцію, яка входить до його складу.

Отже, перевіримо як працює модуль random, який надає доступ до функції randint():

```
import random
for i in range(3):
    print(random.randint(1, 8))
```

Важливо: функція randint() повертає випадкове ціле число з діапазону між двома цілими числами, які передаються у функцію як аргументи.

Виконавши програму, отримаємо три випадкових цілих числа із діапазону від 1 до 8 включно:

```
6
5
7
```

Важливо: оскільки функція randint() знаходиться у модулі random, то ім'я модуля повинне вказуватися у вигляді префікса (через точку) перед іменем функції, щоб мова програмування Python могла визначити, що дану функцію слід шукати у модулі random.

Для імпортування декількох модулів використовують такий синтаксис:

import модуль1, модуль2, модуль3,...

або імпортують так:

import модуль1
import модуль2
import модуль3, модуль4

Важливо: усі інструкції import необхідно розміщувати у верхній частині файлу. Після цього можна використовувати будь-які функції, що зазначені у цих модулях.

Оператор import має альтернативну форму використання:

from модуль import функція1, функція2,...

Слід зазначити, що у такому записі імпортується не весь модуль з повним набором фукнцій, а лише конкретні функції з цього модуля. У даному випадку у коді не потрібно використовувати префікс модуля перед іменем функції:

```
from random import randint, random
for i in range(3):
    print(randint(1, 8))
print(random())
```

2
1
2
0.4003819205039376 (результат виконання функції random())

Важливо: функція random() викликається без аргументів та генерує випадкове дійсне число від 0 до 1, не включаючи межі даного діапазону.

Слід зазначити, що, зазвичай, використовують ім'я функції, перед яким записане ім'я модуля, наприклад, random.randint().

Під час імпортування можна використовувати символ «*»:

from random import *

Слід зазначити, що даний запис можна прочитати так: із модуля random() імпортувати все.

Крім того, для скорочення назв імпортованих модулів (чи функцій із модулів) необхідно користуватися псевдонімами, форма запису їх така:

import модуль as псевдонім (псевдонім для модуля)
from модуль import функція as псевдонім (псевдонім для функції)

Приклад використання псевдоніма rn для модуля random:

```
import random as rn
for i in range(3):
    print(rn.randint(1, 8))
3
5
2
```

Модуль __main__ у мові програмування Python

У файлах мови програмування Python розміщують таку конструкцію:

```
...
if __name__ == '__main__':
    блок коду
```

Для прикладу визначимо функцію у файлі users.py:

```
def greet_users(names):
    '''Виведення привітання для усіх користувачів у списку.'''
    for name in names:
        message = 'Hello, ' + name.title() + '!'
        print(message)
usernames = ['iryna', 'maryna', 'galyna', 'ivan']
greet_users(usernames)
```

Дана функція отримує список імен користувачів і формує вітання усім:

```
Hello, Iryna!
Hello, Maryna!
Hello, Galyna!
Hello, Ivan!
```

Отже, файл users.py – це модуль, що містить одну функцію. Створимо ще один файл main_file.py та помістимо у нього такий код:

```
from users import greet_users
print('This code is executed!')
if __name__ == '__main__':
    print('This code is executed because the main_file.py is not being imported!')
```

Слід зазначити, що у файл main_file.py виконується імпорт функції greet_users() із модуля-файлу users.py. Другий рядок коду виконується під час запуску файлу main_file.py, а також у випадку, коли файл main_file.py імпортуємо у інший файл another_file.py (файл main_file.py стає модулем):

```
import main_file
```

У третьому рядку коду умова if та відповідний блок коду виконуються лише у випадку, коли запукається на виконання сам файл main_file.py.

Результати виконання файлів будуть такі:

```
C:\Python34>python users.py
Hello, Iryna!
Hello, Maryna!
Hello, Galyna!
Hello, Ivan!

C:\Python34>python main_file.py
Hello, Iryna!
Hello, Maryna!
Hello, Galyna!
Hello, Ivan!
This code is executed!
This code is executed because the main_file.py is not being imported!

C:\Python34>python another_file.py
Hello, Iryna!
Hello, Maryna!
Hello, Galyna!
Hello, Ivan!
This code is executed!
```

У першому результаті виконується запуск файлу users.py із функцією, яка повертає результи своєї роботи.

У другому результаті виконується запуск файлу main_file.py, у який імпортували функцію з файлу users.py. Конструкція if __name__ == '__main__': виконується.

У третьому результаті виконується запуск файлу another_file.py, у який імпортували код файлу main_file.py. Конструкція if __name__ == '__main__': не виконується.

Аргументи командного рядка у мові програмування Python

Під час запуску програми на мові програмування Python, яка збережена у файлі з певним ім'ям, у термінальму вікні (вікні командного рядка) необхідно ввести python і ім'я цього файлу.

Створимо файл test.py, який міститиме такі рядки:

```
import sys
print('Program arguments:', sys.argv)
```

Запустимо цей файл у термінальному вікні, попередньо перейшовши у папку, де файл був збережений:

```
C:\Python34>python test.py
Program arguments: ['test.py']

C:\Python34>python test.py tra la la
Program arguments: ['test.py', 'tra', 'la', 'la']
```

Змінна argv з модуля sys має список аргументів командного рядка. Значення argv[0] – ім'я файлу, який запускається (або повний шлях до нього). argv[1], argv[2] і т. д. – це інші аргументи командного рядка.

З прикладу видно, що argv[0] має значення test.py, argv[1] має значення tra, argv[2] має значення la, argv[3] має значення також la.

Пакети у мові програмування Python

Модулі у мові програмування Python ораганізовують у групи файлів, що називаються *пакетами*.

Важливо: пакет – це каталог, який містить файл __init__.py, файли модулів та інші підпакети.

Створення власних пакетів у мові програмування Python

Створимо власний пакет на такому прикладі: необхідно дізнатися прогноз погоди по двом типам (на слідуючий день та на слідуючий тиждень).

Створимо папку boxes, а у ній папку sources, яка буде містити два модуля: файли daily.py і weekly.py, кожний з них міститиме функцію forecast().

Слід зазначити, що версія файлу погоди на кожний день повертатиме рядок, а версія файлу погоди на кожен тиждень повертатиме список із 7 рядків.

Важливо: функція enumerate() розбиває список на частини та відправляє кожний елемент списку в цикл for, додаючи до кожного елемента число – порядковий номер, починаючи з 1.

Основна програма міститься у файлі за адресою boxes\weather.py:

```
from sources import daily, weekly
print('Daily forecast:', daily.forecast())
print('Weekly forecast:')
for number, outlook in enumerate(weekly.forecast(), 1):
        print(number, outlook)
```

Модуль 1 знаходиться у файлі за адресою boxes\sources\daily.py:

```
def forecast():
    'fake daily forecast'
    return 'like yesterday'
```

Модуль 2 знаходиться у файлі за адресою boxes\sources\weekly.py:

```
def forecast():
    '''Fake weekly forecast'''
    return ['mist', 'strong winds', 'sleet', 'freezing rain', 'rain', 'fog', 'drizzle']
```

Важливо: у модулях 1 та 2 присутні коментарі 'fake daily forecast' та '''Fake weekly forecast''' відповідно, створені за допомогою різних типів лапок. Інтерпретатор ігнорує рядки з цими коментарями під час виконання програми.

Слід зазначити, що у папці sources необхідно розмістити файл __init__.py, цей файл може бути порожнім, але він потрібний для того, щоб мова програмування Python могла вважати папку, яка його містить, пакетом.

Виконання основної програми weather.py дасть такий результат:

```
Daily forecast: like yesterday
Weekly forecast:
1 mist
2 strong winds
3 sleet
```

4 freezing rain
5 rain
6 fog
7 drizzle

Використання пакету pip у мові програмування Python

Слід зазначити, що пакет pip – це система управління пакетами та найпопулярніший спосіб встановити сторонні (нестандартні) пакети мови програмування Python.

Важливо: починаючи з версії мови програмування Python 3.4, пакет pip є стандартною частиною Python.

Відкриємо термінальне вікно та введемо команду:

pip3 help

На екрані з'явиться інформація про команди pip та їх використання:
– для оновлення самого pip використовують команду:

pip3 install --upgrade pip

– для перегляду списку установлених пакетів використовують команду:

pip3 list

– для установки пакету використовують команду:

pip3 install назва_пакета

– для видалення пакету використовують команду:

pip3 uninstall назва_пакета

– для оновлення пакету використовують команду:

pip3 install --upgrade назва_пакета

Стандартна бібліотека мови програмування Python

Однією із основних переваг мови програмування Python є те, що вона має велику стандартну бібліотеку модулів, які виконують множину завдань та розташовуються окремо один від одного, щоб уникнути розростання ядра мови.

Перед початком вирішення конкретного завдання необхідно перевірити чи існує стандартний модуль, який реалізовує виконання потрібної функції.

Важливо: мова програмування Python надає документацію для модулів.

Обробка відсутніх ключів словника: функція setdefault()

Слід зазначити, що спроба отримати доступ до словника за допомогою неіснуючого ключа генерує виняток (помилку).

Важливо: функція setdefault() створює елемент словника із ключем, якщо заданий ключ у словнику відсутній:

```
thematic_layer = {‘Roads’: 280, ‘Trees’: 2018}
print(thematic_layer)
quarters = thematic_layer.setdefault(‘Quarters’, 25)
print(thematic_layer)
```

У першому рядку кода виконується визначання словника, у другому – виведення значення ключів та їх значень на екран, у третьому рядку продемонстровано використання функції setdefault(): якщо ключа ще немає у словнику (ключ ‘Quarters’ у словнику відсутній), то у словник буде додано даний ключ з новим значенням «25», у четвертому рядку – виведення результату додавання пари ключ: значення.

Отримаємо такий результат:

```
{‘Trees’: 2018, ‘Roads’: 280}
{‘Trees’: 2018, ‘Quarters’: 25, ‘Roads’: 280}
```

Слід зазначити, якщо необхідно присвоїти інше значення за замовчуванням вже існуючому ключу, то:

```
thematic_layer = {‘Roads’: 280, ‘Trees’: 2018}
print(thematic_layer)
trees = thematic_layer.setdefault(‘Trees’, 2020)
print(thematic_layer)
```

Мова програмування Python поверне оригінальне значення, без змін:

```
{‘Roads’: 280, ‘Trees’: 2018}
{‘Roads’: 280, ‘Trees’: 2018}
```

Підрахунок елементів: функція Counter()

Часто під час розв’язання задач необхідно порахувати кількість елементів послідовності. Якщо говорити про такі лічильники, то в стандартній бібліотеці Python є спеціальна функція Counter(), яка міститься у модулі collections.

```
from collections import Counter
layers = [‘roads’, ‘trees’, ‘lights’, ‘houses’]
layers_counter = Counter(layers)
print(layers_counter)
```

Отримаємо такий результат роботи лічильника:

```
Counter({‘roads’: 1, ‘trees’: 1, ‘lights’: 1, ‘houses’: 1})
```

Упорядкування словника по ключу: функція OrderedDict()

Як відомо, порядок ключів у словнику не можна передбачити.

Розглянемо такий приклад:

```
regions = {
    ‘sands’: ‘Sahara’,
    ‘mountains’: ‘Carpathians’,
    ‘rivers’: ‘Dnieper’,
}
for region in regions:
    print(region)
```

Результатом виведення будуть ключі, але не обов’язково у тому порядку, в якому вони були додані до словника:

```
sands
rivers
mountains
```

Важливо: функція OrderedDict(), яка є у модулі collections, запам’ятовує порядок, в якому додавалися ключі, і повертає їх у тому ж порядку.

Розглянемо приклад OrderedDict із послідовності кортежів виду «ключ: значення», у якому порядок ключів словника збережено під час виведення:

```
from collections import OrderedDict
regions = OrderedDict([
    (‘mountains’, ‘Carpathians’),
    (‘sands’, ‘Sahara’),
    (‘rivers’, ‘Dnieper’)
])
for region in regions:
    print(region)
```

Отримаємо такий результат:

```
mountains
sands
rivers
```

Виведення на екран: функція pprint()

Раніше вже зазначалося, що для виведення інформації на екран використовують функцію print(). У випадку, коли результати виведення важко прочитати, можна використовувати pretty printer (гарний принтер) – функцію pprint(), яка є у модулі pprint:

```
from collections import OrderedDict
from pprint import pprint
regions = OrderedDict([
    (‘mountains’, ‘Carpathians’),
    (‘sands’, ‘Sahara’),
    (‘rivers’, ‘Dnieper’)
])
print(regions)
pprint(regions)
```

Існує можливість порівняти результати використання функції print() та pprint() за допомогою таких виведень:

OrderedDict([(‘mountains’, ‘Carpathians’), (‘sands’, ‘Sahara’), (‘rivers’, ‘Dnieper’)])

```
{‘mountains’: ‘Carpathians’,
‘sands’: ‘Sahara’,
‘rivers’: ‘Dnieper’}
```

3 РОЗРОБКА ДОДАТКІВ З ВИКОРИСТАННЯМ МОВИ ПРОГРАМУВАННЯ PYTHON

3.1 Об’єктно-орієнтований підхід у мові програмування Python

План

1. Особливості об’єктно-орієнтованого програмування та об’єкти мови програмування Python.
2. Створення та використання класу у мові програмування Python.
3. Створення класів та екземплярів на мові програмування Python.
4. Наслідування у мові програмування Python.
5. Перевизначення методу у мові програмування Python.
6. Екземпляри як атрибути у мові програмування Python.

Особливості об’єктно-орієнтованого програмування та об’єкти мови програмування Python

Об’єктно-орієнтоване програмування (ООП) вважається однією із найефективніших методологій створення програмних продуктів.

Слід зазначити, що в ООПпередбачають класи, що описують реально існуючі предмети та ситуації, а потім створюють об’єкти на основі цих описів.

Таким чином, під час створення класу визначається загальна поведінка для цілої категорії об’єктів, а під час створення конкретних об’єктів на базі цих класів, кожен об’єкт автоматично наділяється загальною поведінкою.

Крім того, у кожний об’єкт можна додати унікальні особливості.

Слід зазначити, що у мові програмування Python усе, від чисел до модулів, є об’єктами.

Наприклад, вираз number = 90 є створенням об’єкту типу int із цілочисельним значенням 90, а також присвоєнням посилання на нього за іменем number.

Важливо: об’єкт містить як дані (змінні, які називаються *атрибутами*), так і код (функції, які називаються *методами*). Створення об’єкта на основі класу називається *створенням екземпляра* якогось конкретного предмета.

Наприклад, об’єкт із цілочисельний значенням 90 може використовувати методи додавання та множення. Число 50 – це вже інший цілочисельний об’єкт. Отже, існує клас Integer, якому належать об’єкти 90 та 50.

Крім того, String, наприклад, є вбудованим класом мови програмування Python, який створює рядкові об’єкти: ‘Ukraine’, ‘Poland’. Рядки ‘Ukraine’ та ‘Poland’ мають методи, наприклад, capitalize() та replace().

Мова програмування Python має багато інших вбудованих класів, що дозволяють створювати інші стандартні типи даних, включаючи списки, словники і т. д.

Таким чином, об’єкти можна вважати іменниками, а їх методи – дієсловами. Під об’єктом розуміють окрему річ, під методами визначають, як вона взаємодіє із іншими речами.

Створення та використання класу у мові програмування Python

Наведемо приклад простого класу Thematic_layer, що представляє тематичний шар – не якийсь конкретний, а тематичний шар взагалі. Що можна сказати про тематичні шари? У них є тип та символ. Крім того, відомо, що тематичні шари відображають просторові об’єкти на карті [15].

Два види інформації (тип та символ) та вид поведінки (відображають просторові об’єкти та підписи на карті) будуть включені у клас Thematic_layer, тому що вони є загальними для більшості тематичних шарів.

Клас повідомляє мові програмування Python, як створити об’єкт, який є тематичним шаром. Створивши клас, його використовують для створення екземплярів, кожен з яких представляє один конкретний тематичний шар.

Створення класу

Важливо: функція, що є частиною класу, називається *методом*. Відмінність між функціями та методами у способі виклику методів.

У всіх екземплярах, створених на основі класу Thematic_layer, буде зберігатися назва шару та кількість просторових об’єктів у ньому, а також буде присутній метод display():

```
class Thematic_layer:
    ‘‘‘Тематичний шар.’’’
    def __init__(self, name, number_spatial_objects):
        ‘‘‘Ініціалізація атрибутів name та number_spatial_objects.’’’
        self.name = name
        self.number_spatial_objects = number_spatial_objects

    def display(self):
        ‘‘‘Тематичні шари відображають просторові об’єкти на карті.’’’
        print(self.name.title() + ‘ now display. ’)
```

У даному прикладі спочатку визначається клас із ім’ям Thematic_layer (імена для класів у мові програмування Python починаються з великої літери).

Важливо: якщо суперкласу для даного класу не існує, то після імені класу дужки не ставлять, інакше – у дужках записують ім’я суперкласу.

У другому рядку записано коментар із коротким описом класу.

Метод __init__ – це спеціальний метод, який автоматично виконується під час створення кожного нового екземпляра на базі класу Thematic_layer.

Слід зазначити, що ім’я методу __init__ починається і закінчується двома символами підкреслення. Така схема запису унеможливлює конфлікти імен стандартних методів мови програмування Python та методів власних класів.

У даному прикладі метод __init__ визначається із трьома параметрами, що записуються у дужках: self, name, number_spatial_objects.

Параметр self є обов’язковим у визначенні методу, тому він записаний першим у списку усіх параметрів. Крім того, self є включеним у визначення методу для того, щоб у майбутньому виклику методу __init__ (для створення екземпляра класу Thematic_layer) автоматично передавався аргумент self.

Таким чином, під час кожного виклику методу, пов’язаного із даним класом, автоматично передається self-посилання на екземпляр. Зазначене посилання надає конкретному екземпляру доступ до атрибутів та методів класу.

Слід зазначити, що під час створення екземпляра класу Thematic_layer, мова програмування Python викликає метод __init__ із класу Thematic_layer.

Далі відбувається передача класу Thematic_layer() в аргументах назви шару та кількості просторових об’єктів у ньому, значення self передається автоматично (передавати не потрібно).

Отже, кожного разу, коли необхідно створити екземпляр на основі класу Thematic_layer, слід надавати значення лише двом останнім аргументам name та number_spatial_objects.

Слід зазначити, що кожна із двох змінних має префікс self. Крім того, будь-яка змінна з префіксом self доступна для кожного методу у класі, можна звернутися до цих змінних у кожному екземплярі, створеному на основі класу.

Таким чином, конструкція self.name = name отримує значення, яке знаходиться у параметрі name, та зберігає його в змінну name, яка потім зв’язується зі створюваним екземпляром.

Аналогічним чином працює конструкція self.number_spatial_objects = number_spatial_objects.

Слід зазначити, що змінні, до яких звертаються через экземпляри, теж називаються *атрибутами*.

У класі Thematic_layer також визначається метод: display(). Даному методу не потрібна додаткова інформація (назва шару та кількість просторових об’єктів у ньому), він визначається із єдиним параметром self.

Екземпляри, що будуть створені пізніше, зможуть викликати цей метод.

У даному прикладі метод display() обмежується виведенням повідомлення про те, що тематичний шар відображає просторові об’єкти на карті.

Створення екземпляра класу

Слід зазначити, що клас – це інструкція по створенню екземплярів, тобто клас Thematic_layer – інструкція по створенню екземплярів, які представляють конкретні тематичні шари.

Отже, створимо екземпляр, який представляє конкретний тематичний шар, використовуючи клас Thematic_layer:

```
layer = Thematic_layer(‘trees’, 255)
print(‘Layer’s name is ’ + layer.name.title() + ‘.’)
print(‘Layer is ’ + str(layer.number_spatial_objects) + ‘ spatial objects.’)
```

У даному прикладі створюється екземпляр шару layer із назвою «Trees» та кількістю просторових об’єктів у ньому – 255. У процесі обробки цього рядка мова програмування Python викликає метод __init__ класу Thematic_layer із аргументами ‘trees’ та 255.

Метод __init__ створює екземпляр, який представляє конкретний тематичний шар, та присвоює атрибутам name та number_spatial_objects цього екземпляру передані значення. Метод __init__ не має явної команди return, але мова програмування Python автоматично повертає екземпляр, який представляє тематичний шар. Цей екземпляр зберігається у змінну layer.

Важливо: зазвичай, вважається, що ім’я, яке починається із символу верхнього регістра (наприклад, Thematic_layer), позначає клас, а ім’я, записане у нижньому регістрі (наприклад, layer), позначає окремий екземпляр, створений на основі класу.

Доступ до атрибутів

Важливо: для звернення до атрибутів екземпляра використовується *точковий* запис.

Отже, звертаються до значення атрибута name екземпляра layer так:

```
layer.name
```

Точковий запис часто використовується у мові програмування Python, цей синтаксис показує, як саме Python шукає значення атрибутів.

У даному випадку мова програмування Python звертається до екземпляру layer та шукає атрибут name, що пов'язаний із екземпляром layer, це атрибут, який позначався self.name у класі Thematic_layer.

Аналогічним чином використовується даний прийом для роботи із атрибутом number_spatial_objects.

У першій команді print виклик layer.name.title() записує назву шару 'trees' (значення атрибута name екземпляра layer) з великої літери.

У другій команді print виклик str(layer.number_spatial_objects) перетворює 255, значення атрибута number_spatial_objects екземпляра layer, у рядок.

Даний приклад виводить інформацію про layer:

Layer's name is Trees.
Layer is 255 spatial objects.

Виклик методів

Після створення екземпляра на основі класу Thematic_layer можна застосовувати точковий запис для виклику будь-яких методів, визначених у Thematic_layer.

Щоб викликати метод необхідно вказати екземпляр (у даному випадку layer) та метод, який викликається, розділивши їх крапкою:

```
layer.display()
```

У ході обробки layer.display() мова програмування Python шукає метод display() у класі Thematic_layer та виконує його код:

Trees now display.

Створення декількох екземплярів

На основі класу можна створити множину екземплярів. Наприклад, створимо ще один екземпляр класу Thematic_layer із ім'ям roads:

```
layer_1 = Thematic_layer('trees', 255)
layer_2 = Thematic_layer('roads', 150)

print('Layer's_1 name is ' + layer_1.name.title() + '.')
print('Layer_1 is ' + str(layer_1.number_spatial_objects) + ' spatial objects.')
layer_1.display()
print('Layer's_2 name is ' + layer_2.name.title() + '.')
print('Layer_2 is ' + str(layer_1.number_spatial_objects) + ' spatial objects.')
layer_2.display()
```

У цьому прикладі створюються два екземпляри з іменами Trees та Roads:

```
Layer’s_1 name is Trees.
Layer_1 is 255 spatial objects.
Trees now display.
Layer’s_2 name is Roads.
Layer_2 is 150 spatial objects.
Roads now display.
```

Створення класів та екземплярів на мові програмування Python

Класи можуть використовуватися для моделювання багатьох реальних ситуацій. Після того як клас визначено, створюють екземпляри на основі цього класу. Слід зазначити, що однією із перших задач є зміна атрибутів, пов’язаних із конкретним екземпляром.

Атрибути екземпляра можна змінювати безпосередньо або написати методи, які змінюють атрибути за певними правилами:

```
class Thematic_layer:
    def __init__(self, name, number_spatial_objects):
        self.name = name
        self.number_spatial_objects = number_spatial_objects

    def get_descriptive_name(self):
        ‘‘‘Повертає опис у відформатованому вигляді.’’’
        full_information = self.name + ‘ ’ + str(self.number_spatial_objects)
        return full_information.title()

layer = Thematic_layer(‘trees’, 255)
print(layer.get_descriptive_name())
```

У наведеному класі Thematic_layer визначається метод __init__, його список параметрів починається із self, за яким записано ще два параметри: name та number_spatial_objects. Метод __init__ отримує дані параметри та зберігає їх в атрибутах, які будуть пов’язані з екземплярами, створеними на основі класу.

Під час створення нового екземпляра класу Thematic_layer необхідно вказати назву шару та кількість просторових об’єктів для даного екземпляра.

Визначається метод get_descriptive_name(), який об’єднує кількість просторових об’єктів та назву шару в один рядок з описом, що дозволяє виводити значення кожного атрибута не окремо, а цілим рядком.

Для роботи зі значеннями атрибутів у цьому методі використовується синтаксис self.name, self.number_spatial_objects. Далі створюється екземпляр класу Thematic_layer, який зберігається у змінну layer.

Виклик методу get_descriptive_name() показує, із яким тематичним шаром працює програма:

Trees 255

Присвоювання атрибуту значення за замовчуванням

Кожен атрибут класу повинен мати початкове значення, навіть якщо воно дорівнює 0 або є порожнім рядком. У деяких випадках (наприклад, під час встановлення значень за замовчуванням) це початкове значення необхідно зазначити у тілі методу __init__. Отже, за таких умов передавати параметр для цього атрибута при створенні об'єкта не обов'язково.

Додамо у клас Thematic_layer атрибут, що змінюється з часом – дата останнього редагування тематичного шару, яка оновлюється при векторизації.

Передбачимо у класі атрибут з ім'ям data_reading, форматом якого є 00/00/00. Також у клас буде включено метод read_data() для читання дати останнього редагування тематичного шару:

```
class Thematic_layer:
    def __init__(self, name, number_spatial_objects):
        self.name = name
        self.number_spatial_objects = number_spatial_objects
                self.data_reading = '12/05/18'

    def get_descriptive_name(self):
        '''Повертає опис у відформатованому вигляді.'''
        full_information = self.name + ' ' + str(self.number_spatial_objects)
        return full_information.title()

        def read_data(self):
            '''Виводить дату останнього редагування'''
            print('This thematic layer has ' + str(self.data_reading) + '.')

layer = Thematic_layer('trees', 255)
print(layer.get_descriptive_name())
layer.read_data()
```

Коли мова програмування Python викликає метод __init__ для створення нового екземпляра layer, цей метод зберігає назву шару та кількість просторових об'єктів в атрибутах, як і у попередньому випадку. Потім мова програмування Python створює новий атрибут з ім'ям data_reading і присвоює йому початкове значення '12/05/18' – дату створення шару.

Також у клас додається новий метод read_data(), який спрощує читання дати останнього редагування тематичного шару.

Виклик методів get_descriptive_name() та read_data() для екземпляра layer дає такі початкові дані для нового тематичного шару:

Trees 255
This thematic layer is created 12/05/18.

Зміна значень атрибутів

Значення атрибута у наведеному прикладі можна змінити одним із трьох способів [9]:

– змінити значення безпосередньо в екземплярі;
– задати значення;
– змінити значення за допомогою методу.

Слід зазначити, щоб змінити значення атрибута, найпростіше звернутися до нього безпосередньо через екземпляр. Зазначимо, наприклад, дату останнього редагування тематичного шару такою '15/05/18':

layer.data_reading = '15/05/18'
layer.read_data()

Точковий запис використовується для звернення до атрибуту data_reading екземпляра та прямого присвоєння йому значення.

Рядок layer.data_reading = '15/05/18' вказує мові програмування Python взяти екземпляр layer, знайти пов'язаний з ним атрибут data_reading та задати значення атрибута рівним '15/05/18':

Trees 255
This thematic layer is created 15/05/18.

У клас можна включити методи, які змінюють деякі атрибути автоматично. Замість того, щоб змінювати атрибут безпосередньо, необхідно передати нове значення методу, який оновлює атрибут.

У прикладі наприкінці класу включається метод update_data() для зміни дати останнього редагування тематичного шару:

```
class Thematic_layer:
    def __init__(self, name, number_spatial_objects):
        self.name = name
        self.number_spatial_objects = number_spatial_objects
        self.data_reading = '12/05/18'

    def get_descriptive_name(self):
        full_information = self.name + ' ' + str(self.number_spatial_objects)
        return full_information.title()

    def update_data(self, data):
        '''Встановлює задану дату останнього редагування
        тематичного шару.'''
        self.data_reading = data

    def read_data(self):
        print('This thematic layer is changed ' + str(self.update_data) + '.')

layer = Thematic_layer('trees', 255)
print(layer.get_descriptive_name())
layer.update_data = '15/05/18'
layer.read_data()
```

Клас Thematic_layer майже не змінився, у ньому тільки додався метод update_data(), цей метод отримує дату останнього редагування тематичного шару та зберігає її в self.data_reading.

Потім викликається метод update_data() і йому передається значення '15/05/18' в аргументі (відповідному параметру data у визначенні методу). Метод встановлює дату останнього редагування тематичного шару '15/05/18'.

Метод read_data() доповнює інформацію про дату останнього редагування тематичного шару:

```
Trees 255
This thematic layer is changed 15/05/18.
```

Метод update_data() можна розширити так, щоб під час кожної зміни дати останнього редагування тематичного шару виконувалася деяка додаткова робота.

Додамо перевірку, яка гарантує, що ніхто не буде намагатися змінити дату останнього редагування тематичного шару:

```
class Thematic_layer:
    def __init__(self, name, number_spatial_objects):
        self.name = name
        self.number_spatial_objects = number_spatial_objects
        self.data_reading = '12/05/18'

    def get_descriptive_name(self):
        full_information = self.name + ' ' + str(self.number_spatial_objects)
        return full_information.title()

    def update_data(self, data):
        '''Встановлює задану дату останнього редагування
        тематичного шару. Під час спроби записати дату, що вже минула,
        зміна відхиляється'''
        if data >= self.data_reading:
            self.data_reading = data
        else:
                print('You can not record a date that has already passed!')

    def read_data(self):
        print('This thematic layer is created ' + str(self.data_reading) + '.')

layer = Thematic_layer('trees', 255)
print(layer.get_descriptive_name())
layer.update_data('10/05/18')
layer.read_data()
```

Тепер update_data() перевіряє нове значення перед зміною атрибута. Якщо нове значення data більше або дорівнює поточному self.data_reading, то показники дати останнього редагування тематичного шару можна оновити новим значенням. Якщо ж нове значення менше поточного, то отримаємо попередження про те, що введена дата вже минула:

```
Trees 255
You can not record a date that has already passed!
This thematic layer is created 12/05/18.
```

Слід зазначити, що іноді значення атрибута потрібно змінити на майбутнє прогнозоване значення. Припустимо, необхідно векторизувати у певному тематичному шарі додаткові просторові об'єкти, наприклад, 20/05/18:

```
class Thematic_layer:
    def __init__(self, name, number_spatial_objects):
        self.name = name
        self.number_spatial_objects = number_spatial_objects
        self.data_reading = '12/05/18'

    def get_descriptive_name(self):
        full_information = self.name + ' ' + str(self.number_spatial_objects)
        return full_information.title()

    def update_data(self, data):
        if data >= self.data_reading:
            self.data_reading = data
        else:
            print('You can not record a date that has already passed!')

    def read_data(self):
        print('This thematic layer want to change ' + str(self.data_reading) + '.')

layer = Thematic_layer('trees', 255)
print(layer.get_descriptive_name())
layer.update_data('20/05/18')
layer.read_data()
```

```
Trees 255
This thematic layer want to change 20/05/18.
```

Наслідування у мові програмування Python

Робота над новим класом не обов'язково повинна починатися з нуля. Якщо клас, який заплановано написати, є спеціалізованою версію раніше написаного класу, то можна скористатися *наслідуванням* (успадкуванням) [2].

Клас, що застосовує принцип наслідування, автоматично отримує всі атрибути та методи вихідного класу.

Вихідний клас називається *батьком*, а новий клас на його основі – *нащадком*. Клас-нащадок успадковує атрибути та методи батька, але при цьому також може визначати власні атрибути та методи [1].

Перше, що робить мова програмування Python під час створення екземпляра класу-нащадка, – додає значення всіх атрибутів класу-батька.

Щоб це реалізувати методу __init__ класу-нащадка необхідна допомога з боку класу-батька.

Наприклад, створимо тематичний шар кварталів.

Тематичний шар кварталів є спеціалізованим різновидом тематичного шару, тому новий клас Thematic_layer_of_the_quarters (нащадок) можна створити на базі класу Thematic_layer (батько), написаного раніше. Необхідно додати в нього код атрибутів та поведінки, що відноситься тільки до тематичного шару кварталів:

```
class Thematic_layer:
    def __init__(self, name, number_spatial_objects):
        self.name = name
        self.number_spatial_objects = number_spatial_objects
        self.data_reading = '12/05/18'

    def get_descriptive_name(self):
        full_information = self.name + ' ' + str(self.number_spatial_objects)
        return full_information.title()

    def update_data(self, data):
        if data >= self.data_reading:
           self.data_reading = data
        else:
              print('You can not record a date that has already passed!')

    def read_data(self):
        print('This thematic layer is created ' + str(self.update_data) + '.')

class Thematic_layer_of_the_quarters(Thematic_layer):
    def __init__(self, name, number_spatial_objects):
        super().__init__(name, number_spatial_objects)

layer_of_the_quarters = Thematic_layer_of_the_quarters('microdistrict-531', 50)
print(layer_of_the_quarters.get_descriptive_name())
```

Створюється екземпляр Thematic_layer. Під час створення класу-нащадка клас-батько повинен бути частиною поточного файлу, а його визначення повинно передувати визначенню класу-нащадка у файлі.

Визначається клас-нащадок Thematic_layer_of_the_quarters. У визначенні нащадка ім’я класу-батька записується у круглих дужках.

Слід зазначити, що метод __init__ отримує інформацію, яка необхідна для створення екземпляра Thematic_layer.

Крім того, функція super() – спеціальна функція, яка допомагає мові програмування Python зв’язати нащадка з батьком. Даний рядок надає команду мові програмування Python викликати метод __init__ класу, який є батьком Thematic_layer_of_the_quarters, екземпляр класу Thematic_layer_of_the_quarters отримує всі атрибути класу-батька. Ім’я super виходить із такої термінології: клас-батько називається *суперкласом*, а клас-нащадок – *субкласом*.

Щоб перевірити чи правильно спрацювало наслідування, спробуємо створити тематичний шар кварталів із такою ж інформацією, яка передається під час створення звичайного екземпляра Thematic_layer.

Створюємо екземпляр класу Thematic_layer_of_the_quarters та зберігаємо його у змінну layer_of_the_quarters. Даний рядок викликає метод __init__, який визначений у класі Thematic_layer_of_the_quarters, що, у свою чергу, вказує мові програмування Python викликати метод __init__, визначений у класі-батька Thematic_layer. Під час виклику передаються аргументи ‘microdistrict-531’ та 50.

Крім __init__, клас не містить ніяких інших атрибутів або методів, специфічних для тематичного шару кварталів. Тобто клас тематичного шару кварталів містить всю поведінку, властиву класу звичайного тематичного шару:

Microdistrict-531 55

Після створення класу-нащадка, який успадковує все від класу-батька, можна переходити до додавання нових атрибутів та методів, необхідних для того, щоб нащадок відрізнявся від батька.

Додамо атрибут, який можна вважати специфічним для тематичного шару кварталів (наприклад, площа), та метод для виведення інформації про цей атрибут:

```
class Thematic_layer:
    def __init__(self, name, number_spatial_objects):
        self.name = name
        self.number_spatial_objects = number_spatial_objects
        self.data_reading = ‘12/05/18’
```

```
    def get_descriptive_name(self):
        full_information = self.name + ‘ ’ + str(self.number_spatial_objects)
        return full_information.title()

class Thematic_layer_of_the_quarters(Thematic_layer):
    def __init__(self, name, number_spatial_objects):
        super().__init__(name, number_spatial_objects)
        self.area = 0.8

    def describe_area(self):
        print(‘This thematic layer has a quarter area ’ + \
              str(self.area) + ‘ kilometers square.’)

layer_of_the_quarters = Thematic_layer_of_the_quarters \
                        (‘microdistrict-531’, 55)
print(layer_of_the_quarters.get_descriptive_name())
layer_of_the_quarters.describe_area()
```

Додається новий атрибут self.area, якому присвоюється початкове значення – 0.8, цей атрибут буде присутнім у всіх екземплярах, створених на основі класу Thematic_layer_of_the_quarters (але не у всякому екземплярі Thematic_layer).

Крім того, додається метод із ім’ям describe_area(), який виводить інформацію про площу певного кварталу в тематичному шарі кварталів.

Під час виклику методу describe_area() виводиться опис, який відноситься тільки до певного кварталу в тематичному шарі кварталів:

```
Microdistrict-531 55
This thematic layer has a quarter area 0.8 kilometers square.
```

Можливості спеціалізації класу Thematic_layer_of_the_quarters безмежні. Можна додати скільки завгодно атрибутів та методів, щоб моделювати квартал із будь-якими властивостями.

Атрибути або методи, які можуть належати будь-якому тематичному шару (а не тільки тематичному шару кварталів), повинні додаватися до класу Thematic_layer. Дана інформація буде доступною всім користувачам класу Thematic_layer.

Клас Thematic_layer_of_the_quarters буде містити код інформації та поведінку, що специфічні для тематичних шарів кварталів.

Перевизначення методу у мові програмування Python

Будь-який метод батьківського класу, який в конкретній ситуації робить не те, що від нього вимагається, можна перевизначити [2].

Для цього необхідно в класі-нащадку визначити метод із тим же ім'ям, що і у методі класу-батька. Мова програмування Python ігнорує метод у класі батька та звертає увагу тільки на метод, визначений у нащадку.

Наприклад, у класі Thematic_layer є метод vectorize_the_river().

Для кварталів мала ймовірність наявності річки, тому цей метод логічно перевизначити, наприклад:

```
class Thematic_layer:
    def __init__(self, name, number_spatial_objects):
        self.name = name
        self.number_spatial_objects = number_spatial_objects
        self.data_reading = '12/05/18'

    def get_descriptive_name(self):
        full_information = self.name + ' ' + str(self.number_spatial_objects)
        return full_information.title()

    def vectorize_the_river(self):
        print('This thematic layer has rivers.')

class Thematic_layer_of_the_quarters(Thematic_layer):
    def __init__(self, name, number_spatial_objects):
        super().__init__(name, number_spatial_objects)
        self.area = 0.8

    def describe_area(self):
        print('This thematic layer has a quarter area ' + \
              str(self.area) + ' kilometers square.')

    def vectorize_the_river(self):
        print('This thematic layer has no rivers!')

layer = Thematic_layer('trees', 255)
print(layer.get_descriptive_name())
layer.vectorize_the_river()
```

```
layer_of_the_quarters = Thematic_layer_of_the_quarters\
                        (‘microdistrict-531’, 55)
print(layer_of_the_quarters.get_descriptive_name())
layer_of_the_quarters.vectorize_the_river()
```

Якщо викликати метод vectorize_the_river() для звичайного тематичного шару, то мова програмування Python викличе метод vectorize_the_river() класу Thematic_layer.

Якщо хтось спробує викликати метод vectorize_the_river() для тематичного шару кварталів, то мова програмування Python проігнорує метод vectorize_the_river() класу Thematic_layer та виконає замість нього код із метода vectorize_the_river() класу Thematic_layer_of_the_quarters:

```
Trees 255
This thematic layer has rivers.
Microdistrict-531 55
This thematic layer has no rivers!
```

Важливо: із застосуванням наслідування нащадок зберігає аспекти батька, які йому необхідні, та відкидає все непотрібне.

Екземпляри як атрибути у мові програмування Python

Слід зазначити, що під час моделювання явищ реального світу класи доповнюють великою кількістю подробиць, списки атрибутів та методів зростають і через певний час файли стають громіздкими.

У таких ситуаціях частину одного класу можна записати у вигляді окремого класу, тобто великий код розбивається на менші класи, які працюють у взаємодії один із одним.

Під час подальшої роботи із класом Thematic_layer_of_the_quarters може виявитися, що у ньому з’явилося занадто багато атрибутів та методів, що відносяться до площі. У такому випадку можна зупинитися та перемістити усі атрибути та методи в окремий клас із ім’ям Area. Таким чином, екземпляр Area стає атрибутом класу Thematic_layer_of_the_quarters:

```
class Thematic_layer:
    def __init__(self, name, number_spatial_objects):
        self.name = name
        self.number_spatial_objects = number_spatial_objects
        self.data_reading = ‘12/05/18’
```

```
    def get_descriptive_name(self):
        full_information = self.name + ‘ ’ + str(self.number_spatial_objects)
        return full_information.title()

    def vectorize_the_river(self):
        print(‘This thematic layer has rivers.’)

class Area:
    def __init__(self, area_size=0.8):
        self.area_size = area_size

    def describe_area(self):
        print(‘This thematic layer has a quarter area ’ + \
              str(self.area_size) + ‘ kilometers square.’)

class Thematic_layer_of_the_quarters(Thematic_layer):
    def __init__(self, name, number_spatial_objects):
        super().__init__(name, number_spatial_objects)
        self.area = Area()

    def vectorize_the_river(self):
        print(‘This thematic layer has no rivers.’)

layer_of_the_quarters = Thematic_layer_of_the_quarters\
                        (‘microdistrict-531’, 55)
print(layer_of_the_quarters.get_descriptive_name())
layer_of_the_quarters.area.describe_area()
```

У даному прикладі визначається новий клас із ім’ям Area, який не буде наслідувати жодного класу. Метод __init__ отримує self та один параметр за замовчуванням area_size. Отже, якщо значення не вказується користувачем, то цей параметр за замовчуванням задає area_size значення 0.8.

Крім того, метод describe_area() необхідно також перемістити у клас Area.

Потім у клас Thematic_layer_of_the_quarters додається атрибут із ім’ям self.area, що вказує мові програмування Python створити новий екземпляр Area (зі значенням area_size за замовчуванням, рівним 0.8, так як значення не задано) і зберегти його в атрибуті self.area. Даний факт буде відбуватися під час кожного виклику __init__ і будь-який екземпляр Thematic_layer_of_the_quarters матиме автоматично створений екземпляр Area.

Програма створює екземпляр тематичного шару кварталів та зберігає його у змінній layer_of_the_quarters.

Якщо необхідно буде вивести опис площі, то слід звернутися до екземпляра layer_of_the_quarters, знайти атрибут area та викликати метод describe_area(), який пов'язаний із екземпляром Area з атрибуту:

Microdistrict-531 55
This thematic layer has a quarter area 0.8 kilometers square.

Новий варіант програми вимагає великої додаткової роботи, але тепер площу можна моделювати із будь-яким ступенем деталізації без збільшення громіздкості класу layer_of_the_quarters.

Додамо у клас Area ще один метод, який виводить ширину та довжину певного кварталу із тематичного шару кварталів:

```
class Thematic_layer:
    def __init__(self, name, number_spatial_objects):
        self.name = name
        self.number_spatial_objects = number_spatial_objects
        self.data_reading = '12/05/18'

    def get_descriptive_name(self):
        full_information = self.name + ' ' + str(self.number_spatial_objects)
        return full_information.title()

    def vectorize_the_river(self):
        print('This thematic layer has rivers.')

class Area:
    def __init__(self, area_size=0.8):
        self.area_size = area_size

    def describe_area(self):
        print('This thematic layer has a quarter area ' + \
                str(self.area_size) + ' kilometers square.')

    def get_width_and_length(self):
        if self.area_size ==0.8:
            width_and_length = 1, 0.8
```

```
            message = ‘This quarter has a width and length ’ + /
                        str(width_and_length) + ‘ kilometers in accordance.’
            print(message)

class Thematic_layer_of_the_quarters(Thematic_layer):
    def __init__(self, name, number_spatial_objects):
        super().__init__(name, number_spatial_objects)
        self.area = Area()

    def vectorize_the_river(self):
        print(‘This thematic layer has no rivers.’)

layer_of_the_quarters = Thematic_layer_of_the_quarters\
                        (‘microdistrict-531’, 55)
print(layer_of_the_quarters.get_descriptive_name())
layer_of_the_quarters.area.describe_area()
layer_of_the_quarters.area.get_width_and_length()
```

Новий метод get_width_and_length() проводить простий аналіз, якщо площа кварталу 0,8 квадратних кілометрів, то його ширина та довжина може мати такі значення – 1 та 0,8 кілометрів відповідно. Програма виводить дану інформацію.

Отже, якщо необхідно використати метод get_width_and_length(), то його слід викликати через атрибут area.

Результат спрацювання програми повідомляє про ширину та довжину кварталу у кілометрах у залежності від його площі:

```
Microdistrict-531 55
This thematic layer has a quarter area 0.8 kilometers square.
This quarter has a width and length (1, 0.8) kilometers in accordance.
```

Таким чином, використання класів у програмах надає можливість подати особливості реального світу, а також продемонструвати мислення на високому логічному рівні, не обмежуючись рівнем звичайного синтаксису.

3.2 Програмування додатків баз даних на мові програмування Python

План

1. Групи файлів та особливості роботи з ними.
2. Бази даних у мові програмування Python.
3. Особливості роботи із датою та часом у мові програмування Python.

Групи файлів та особливості роботи з ними

Слід зазначити, що активна програма працює з даними, які зберігаються в оперативній пам’яті.

RAM (Random Access Memory) – швидка пам’ять, що вимагає постійного живлення (якщо живлення зникає, то всі дані, які в ній зберігаються, будуть втрачені). На відміну від RAM, різноманітні сучасні накопичувачі можуть зберігати дані навіть після того, як вимкнути живлення [1].

Слід зазначити, що найпростіший приклад сховища даних – це звичайний файл. Дані зчитуються з файлу в пам’ять та записуються з пам’яті у файл.

Серед типів файлів можна виділити дві групи [1]:

– прості текстові файли;

– бінарні файли.

Важливо: прості текстові файли містять лише базові текстові символи, без інформації про шрифт, розмір та колір тексту. Прикладом таких файлів є файли з розширенням .txt або файли Python з розширенням .py.

Важливо: бінарні (двійкові) файли – це файли документів, що створені текстовими процесорами (наприклад, PDF-файли, файли зображень).

Відкриття файлу

Щоб записати інформацію у файл або зчитати з нього дані необхідно попередньо його відкрити:

```
fileobj = open(filename, mode)
```

Під елементами цього запису слід розуміти [1]:

– fileobj – об’єкт файлу, який повертається функцією open();

– filename – рядок з назвою файлу;

– mode – рядок, який вказує на тип файлу та дії, які можна виконувати над файлом.

Крім того, перша літера рядка mode вказує на певну операцію із файлом:

– r – читання з файлу;

– w – запис в існуючий файл (якщо файла не існує, він буде створений);

– x – запис у файл, якщо файл не існує;

– a – додавання у кінець файлу, якщо файл існує.

Друга літера рядка mode вказує на тип файлу:

– t (або нічого) – означає, що файл текстовий;

– b – означає, що файл бінарний.

Таким чином, після відкриття файлу, виконують читання даних із файлу або записують дані у файл. Наприкінці роботи необхідно файл закрити.

Запис даних у текстові файли

Нехай є текстовий рядок, що міститься у змінній text, запишимо його у файл result:

```
text = ‘I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.’
frecord = open(‘result.txt’, ‘wt’)
a = frecord.write(text)
frecord.close()
print(a)
```

Функція write(), значення якої присвоєно змінній *a*, повертає число байтів, що записані у файл:

```
116
```

Однак, функція write() не додає ніяких інших символів у файл, як це робить функція print(), що також дозволяє записувати у файл:

```
text = ‘I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.’
frecord = open(‘result.txt’, ‘wt’)
print(text, file=frecord)
frecord.close()
```

Слід зазначити, що для запису у файл можна використовувати обидві функції, однак, слід пам’ятати про два аргументи функції print() [1]:

– sep – розділювач, за замовчуванням це пропуск (‘ ’);

– end – означає символ кінця файлу, за замовчуванням це символ нового рядка (\n).

Отже, для того, щоб функція print() працювала як функція write(), змінемо код таким чином:

```
text = ‘I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.’
frecord = open(‘result.txt’, ‘wt’)
print(text, file=frecord, sep=‘’, end=‘’)
frecord.close()
```

На прикладі перевіримо, як спрацює режим x щодо файлу result.txt стосовно його перезаписування:

```
text = ‘I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.’
frecord = open(‘result.txt’, ‘xt’)
print(text, file=frecord, sep=‘’, end=‘’)
frecord.close()
```

```
Traceback (most recent call last):
  File “C:\Python34\text.py”, line 2, in <module>
    frecord = open(‘result.txt’, ‘xt’)
FileExistsError: [Errno 17] File exists: ‘result.txt’
```

Отже, перезаписати не вдалося. Виникла помилка і згенерувався виняток. У такому випадку корисно використовувати обробник винятку:

```
text = ‘I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.’
try:
    frecord = open(‘result.txt’, ‘xt’)
    frecord.write(text)
except FileExistsError:
    print(‘result.txt already exists!’)
```

Результатом використання обробника винятку є повідомлення про те, що файл з таким ім’ям вже існує і записати у нього інформацію неможливо:

```
result.txt already exists!
```

Зчитування даних з текстових файлів

Важливо: щоб прочитати дані з файлу, можна використовувати такі функції: read(), readline(), readlines(). Крім того, щоб прочитати увесь вміст файлу за один раз, використовують функцію read().

Важливо: у випадку великих файлів та використання функції read(), для зчитування даних з файлів, необхідний більший обсяг оперативної пам'яті:

```
text = 'I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.'
freading = open('result.txt', 'rt')
text = freading.read()
freading.close()
print(text)
print(len(text))
```

Результат зчитування даних із файлу буде таким – виведення усього вмісту файлу за один раз та виведення кількості байтів, зчитаних з файлу:

```
I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.
116
```

Також можна вказати максимальну кількість символів, які зчитуються з файлу за один раз.

Наприклад, зчитаємо по 20 символів за один раз та приєднаємо кожний фрагмент до рядка text, щоб відтворити увесь вміст файлу:

```
text = 'I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.'
freading = open('result.txt', 'rt')
chunk = 20
while True:
    fragment = freading.read(chunk)
    if not fragment:
        break
    text += fragment
    print(len(fragment))
freading.close()
```

Слід зазначити, що після того як зчитали увесь файл, подальші виклики функції read() будуть повертати порожній рядок (‘ ’), що сприймається як False у перевірці if not fragment та дозволяє вийти із нескінченного циклу while True:

```
20
20
20
20
20
16
```

Крім того, можна зчитувати файл по одному рядку за раз за допомогою функції readline(), приєднаємо кожен рядок до рядка text, щоб відтворити увесь вміст файлу:

```
text = ‘I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.’
freading = open(‘result.txt’, ‘rt’)
while True:
    line = freading.readline()
    if not line:
        break
    text += line
freading.close()
print(text)
```

I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.

Слід зазначити, що для текстового файлу навіть порожній рядок має довжину, яка дорівнює 1 (символ нового рядка \n), такий рядок буде вважатися True. Коли увесь файл буде зчитано, то функція readline() (так само як і функція read()) поверне порожній рядок та сприйме подальшу роботу як False.

Слід зазначити, що найпростіший спосіб прочитати текстовий файл – використати цикл, який буде повертати по одному рядку за один раз.

Даний прийом дозволяє оптимізувати попередньо створений файл.

```
text = ‘I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.’
freading = open(‘result.txt’, ‘rt’)
for line in freading:
    text += line
freading.close()
print(text)
```

I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.

У всіх попередніх прикладах у результаті повертався один рядок text.

Слід зазначити, що існує функція readlines(), яка зчитує по одному рядку за один раз та повертає список цих рядків:

```
text = ‘I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.’
freading = open(‘result.txt’, ‘rt’)
lines = freading.readlines()
freading.close()
print(len(lines), ‘lines read’)
for line in lines:
    print(line, end=‘’)
```

1 lines read

I say thank you to heaven that I breathe and follow my dreams, for the world in which I live, for the people I love.

Бінарні файли

Якщо включити символ ‘b’ у рядок mode, то файл буде відкрито у бінарному режимі, тобто у цьому випадку замість читання та запису рядків операції виконуються із байтами.

Таким чином, щоб продемонструвати роботу із бінарними файлами, згенеруємо 256 байтових значень від 0 до 255 та відкриємо файл bfile у бінарному режимі для запису цих згенерованих даних:

```
bdata = bytes(range(0, 256))
frecord = open(‘bfile’, ‘wb’)
frecord.write(bdata)
frecord.close()
```

Аналогічно до тексту, бінарні дані можна записувати (зчитувати) фрагментами. Запишемо у файл згенеровані бінарні дані фрагментами не більшими за 70 байт:

```
bdata = bytes(range(0, 256))
frecord = open('bfile', 'wb')
size = len(bdata)
offset = 0
chunk = 70
while True:
    if offset > size:
        break
    result = frecord.write(bdata[offset:offset+chunk])
    print(result)
    offset += chunk
frecord.close()
```

Функція print(result) буде візуалізувати цей процес запису – на екран виведеться кількість записаних байтів:

```
70
70
70
46
```

Таким чином, щоб прочитати бінарний файл – необхідно відкрити його у бінарному режимі rb:

```
bdata = bytes(range(0, 256))
freading = open('bfile', 'rb')
bdata = freading.read()
freading.close()
```

Слід зазначити, що для відслідковування місцезнаходження у файлі конкретних байтів використовують функції tell() та seek():

```
bdata = bytes(range(0, 256))
freading = open('bfile', 'rb')
print(freading.tell())
print(freading.seek(25))
```

Функція tell() повертає ціле число – поточне розташування від початку файлу, виміряне у байтах.

Функцію seek() використовують для того, щоб зміститися до конкретного байту у файлі.

```
0
25
```

Крім того, можна викликати функцію seek(), передавши їй другий аргумент seek(offset, origin) або seek(зміщення, походження):

– якщо значення origin дорівнює 0 (за замовчуванням), то зміститися на offset байт із початку файлу;

– якщо значення origin дорівнює 1, то зміститися на offset байт з поточної позиції праворуч;

– якщо значення origin дорівнює 2, то зміститися на offset байт з кінця файлу ліворуч.

Отже, існує можливість перейти до будь-якого байту у файлі.

Продемонструємо вищезазначене за допомогою прикладу:

bdata = bytes(range(0, 256)) – згенеруємо 256 байтових значень.

freading = open('bfile', 'rb') – відкриваємо файл у бінарному режимі.

freading.seek(251, 0) – переходимо у позицію за 5 байтів до кінця файлу.

print(freading.tell()) – виводимо на екран поточну позицію, рахуючи від початку файлу.

freading.seek(3, 1) – переходимо на 3 байти праворуч з поточної позиції.

print(freading.tell()) – виводимо на екран поточну позицію, рахуючи від початку файлу.

bdata = freading.read() – зчитуємо усі байти до кінця файлу.

print(bdata) – виводимо на екран зчитані дані.

Результатами виведення трьох функцій print() у тому ж порядку, як вони записані, будуть такі дані:

```
251
254
b'\xfe\xff'
```

Важливо: функції tell() та seek() найбільш корисні під час роботи із бінарними файлами.

Закриття файлів автоматично з with

Важливо: файл повинен бути закритий після того, як усі операції з ним завершені.

Слід зазначити, що у мові програмування Python є *менеджери контексту* для очищення об'єктів, наприклад, відкритих файлів.

Менеджер контексту має таку конструкцію:

with вираз as змінна:

Наприклад:

```
with open('result.txt', 'wt') as freading:
        freading.write(text)
```

Після того як блок коду, розташований у менеджері контексту (рядок freading.write(text)), завершиться (звичайним способом, або шляхом генерації винятку), файл буде закрито автоматично.

Структуровані текстові файли

Слід зазначити, що для простих текстових файлів єдиним рівнем організації є рядок. Однак, може знадобитися більш структурований файл, у якому необхідно зберегти дані своєї програми для подальшого використання або відправити їх іншій програмі.

На сьогодні існує ряд популярних форматів, які можна розрізнити за такими ознаками:

– розділювач у вигляді табуляції (\t), коми (,) або вертикальної риски (|), це приклад формату *зі значеннями, розділеними комами* (CSV);

– символи < і >, які описують теги, це приклади XML та HTML файлів;

– розділові знаки, прикладом є JavaScript Object Notation (JSON).

Бази даних у мові програмування Python

Базами даних у комп'ютерному світі користуються повсякчас.

Важливо: словосполучення «*база даних*» використовується у декількох випадках: коли мова йде про сервер, про сховище та про дані, які там зберігаються.

Важливо: бази даних називаються *реляційними*, якщо вони показують відношення між різними типами даних, що подані у вигляді таблиць.

Реляційні бази даних мають ряд переваг [1]:

– доступ до даних можливий для декількох користувачів одночасно;

– діє захист від пошкодження даних користувачами;

– існують ефективні методи збереження та зчитування даних;

– об’єднання дозволяють знайти відношення між різними типами даних;

– використання декларативної (структурованої) мови запитів SQL;

– таблиця має вигляд сітки із *записами* (рядками) та *полями* (стовпцями);

– щоб створити таблицю, необхідно вказати її ім’я, імена та типи її полів;

– *первинним ключем* таблиці є поле або група полів, значення яких повинні бути унікальними, що запобігає введення однакових даних у таблицю;

– первинний ключ індексується для більш швидкого пошуку під час виконання запитів. Робота індексу схожа на алфавітний покажчик, що дозволяє швидко знайти певний рядок;

– кожна таблиця знаходиться усередині батьківської бази даних, що нагадує файли в каталозі.

Програмний інтерфейс програми (API)

Важливо: програмний інтерфейс програми (Application Programming Interface) – це набір функцій, що надає доступ до будь-якої послуги.

У мові програмування Python є стандартний API, що призначений для отримання доступу до реляційних баз даних, із його допомогою можна написати програму, яка працює з декількома видами реляційних баз даних.

Розглянемо основні функції стандартного API:

– connect() – реалізація з’єднання з базою даних, виклик цієї функції може включати в себе аргументи, наприклад, ім’я користувача, пароль;

– cursor() – реалізація об’єкта курсору, призначеного для роботи із запитами;

– execute() та executemany() – запуск однієї або більше команд SQL;

– fetchone(), fetchmany() та fetchall() – отримання результатів роботи функції execute().

Мова запитів SQL

Слід зазначити, SQL – це декларативна універсальна мова реляційних баз даних. Запити SQL є текстовими рядками, які комп’ютер (клієнт) відправляє серверу бази даних, а той, у свою чергу, визначає, що з ними робити далі.

Існують дві основні категорії SQL [1]:

– DDL (Data Definition Language, мова визначення даних) – обробляє створення, видалення, обмеження та дозволи для таблиць баз даних (табл. 3.1);

– DML (Data Manipulation Language, мова маніпулювання даними) – обробляє додавання даних, їх вибірку, оновлення та видалення (табл. 3.2).

Важливо: мова SQL не залежить від регістру, але ключові слова пишуться ВЕЛИКИМИ ЛІТЕРАМИ, щоб можна було відрізнити їх від назв полів.

Таблиця 3.1 – Основні команди SQL DDL

Операція	Шаблон SQL	Приклад SQL
Створення бази даних	CREATE DATABASE назва_бази_даних	CREATE DATABASE dbname
Вибір поточної бази даних	USE назва_бази_даних	USE dbname
Видалення бази даних та її таблиць	DROP DATABASE назва_бази_даних	DROP DATABASE dbname
Створення таблиці	CREATE TABLE назва_таблиці (назви полів та їхніх типів)	CREATE TABLE tbname (id INT, count INT)
Видалення таблиці	DROP TABLE назва_таблиці	DROP TABLE tbname
Видалення усіх записів (рядків) таблиці	TRUNCATE TABLE назва_таблиці	TRUNCATE TABLE tbname

Основні операції DML реляційної бази даних можна запам’ятати за допомогою акроніма CRUD [1]:

– Create – створення за допомогою оператора INSERT;

– Read – читання за допомогою SELECT;

– Update – оновлення за допомогою UPDATE;

– Delete – видалення за допомогою DELETE.

Таблиця 3.2 – Основні команди SQL DML

Операція	Шаблон SQL	Приклад SQL
Додавання запису (рядка) у таблицю	INSERT INTO назва_таблиці VALUES(значення)	INSERT INTO tbname VALUES(7, 40)
Вибір усіх записів (рядків) та полів (стовпців) із таблиці	SELECT * FROM назва_таблиці	SELECT * FROM tbname
Вибір всіх записів (рядків) та деяких полів (стовпців) із таблиці	SELECT назва_поля1, назва_поля2 FROM назва_таблиці	SELECT id, count FROM tbname
Вибір деяких записів (рядків) та деяких полів (стовпців) із таблиці	SELECT назва_поля1, назва_поля2 FROM назва_таблиці WHERE назва_поля2 > значення2 AND назва_поля1 = значення1	SELECT id, count FROM tbname WHERE count > 5 AND id = 9
Зміна деяких записів (рядків) у полі (стовпці) таблиці	UPDATE назва_таблиці SET назва_поля1 = значення1 WHERE назва_поля2 = значення2	UPDATE tbname SET count = 3 WHERE id = 5
Видалення деяких записів (рядків) таблиці	DELETE FROM назва_таблиці WHERE назва_поля1 <= значення1 OR назва_поля2 = значення2	DELETE FROM tbname WHERE count <= 10 OR id = 16

Реляційна база даних з відкритим вихідним кодом SQLite

SQLite – це легка реляційна база даних з відкритим вихідним кодом.

SQLite реалізована як стандартна бібліотека мови програмування Python та зберігає бази даних у звичайних файлах. Ці файли можна переносити на інші комп'ютери та в операційні системи, що робить SQLite портативним рішенням для створення простих реляційних баз даних.

SQLite підтримує мову запитів SQL та дозволяє кільком користувачам працювати з нею одночасно [1].

Важливо: браузери та операційні системи використовують SQLite як вбудовану базу даних.

Робота з базою даних починається з виклику connect() для встановлення з'єднання із локальним файлом бази даних, який необхідно створити або використати.

Для прикладу, створимо базу даних ishop.db та таблицю computers, яка буде містити інформацію про товари інтернет-магазину комп'ютерної техніки.

У таблиці будуть міститися такі поля:

– id – унікальний номер товару (первинний ключ);

– name – назва товару (рядок змінної довжини);

– count – кількість одиниць конкретного товару (ціле число);

– price – ціна одного екземпляру конкретного товару (дійсне число).

```
import sqlite3
conn = sqlite3.connect('ishop.db')
curs = conn.cursor()
curs.execute('''CREATE TABLE computers (id INT PRIMARY KEY, name VARCHAR(20), count INT, price FLOAT)''')
conn.commit()
```

Важливо: необхідно використовувати потрійні лапки (''' ''') під час створення довгих рядків та під час створення запитів SQL.

Важливо: якщо записати для поля id тип INTEGER PRIMARY KEY, то значення id буде збільшуватися на одиницю автоматично.

Рядок conn.commit() дозволяє зберети поточні зміни. Перед тим, як завершити роботу із SQLite, необхідно закрити курсор та з'єднання:

```
import sqlite3
conn = sqlite3.connect('ishop.db')
curs = conn.cursor()
```

```
curs.execute(‘‘‘CREATE TABLE computers (id INT PRIMARY KEY, name VARCHAR(20), count INT, price FLOAT)’’’)
conn.commit()
curs.close()
conn.close()
```

Додамо у базу даних магазину декілька товарів:

```
import sqlite3
conn = sqlite3.connect(‘ishop.db’)
curs = conn.cursor()
curs.execute(‘INSERT INTO computers VALUES(1, “PC”, 5, 7570.50)’)
curs.execute(‘INSERT INTO computers VALUES(2, “Notebook”, 8, 11430.30)’)
conn.commit()
curs.close()
conn.close()
```

Слід зазначити, що існує ще один спосіб додавання даних – використання заповнювача у вигляді «?»:

```
import sqlite3
conn = sqlite3.connect(‘ishop.db’)
curs = conn.cursor()
ins = ‘INSERT INTO computers (id, name, count, price) VALUES(?, ?, ?, ?)’
curs.execute(ins, (3, ‘TabletPC’, 4, 3970.20))
ins = ‘INSERT INTO computers (id, name, count, price) VALUES(?, ?, ?, ?)’
curs.execute(ins, (4, ‘Console’, 2, 16780.90))
conn.commit()
curs.close()
conn.close()
```

У запиті використано чотири знаки «?», щоб показати, що заплановано додати чотири значення та потім занести значення списком у функцію execute().

Важливо: заповнювачі полегшують розставлення лапок та захищають від SQL-ін’єкцій (зовнішня атака, поширена у мережі Інтернет, яка впроваджує у систему шкідливі команди SQL).

Перевіримо на прикладі отримання списку товарів:

```
import sqlite3
conn = sqlite3.connect('ishop.db')
curs = conn.cursor()
curs.execute('SELECT * FROM computers')
conn.commit()
rows = curs.fetchall()
print(rows)
curs.close()
conn.close()
```

Отримаємо:

[(1, 'PC', 5, 7570.5), (2, 'Notebook', 8, 11430.3), (3, 'TabletPC', 4, 3970.2), (4, 'Console', 2, 16780.9)]

Упорядкуємо отриманий список за кількістю товарів:

```
import sqlite3
conn = sqlite3.connect('ishop.db')
curs = conn.cursor()
curs.execute('SELECT * FROM computers ORDER BY count')
rows = curs.fetchall()
print(rows)
curs.close()
conn.close()
```

Отримаємо упорядкування за зростанням по полю кількість:

[(4, 'Console', 2, 16780.9), (3, 'TabletPC', 4, 3970.2), (1, 'PC', 5, 7570.5), (2, 'Notebook', 8, 11430.3)]

Упорядкуємо отриманий список за кількістю товарів за спаданням:

```
import sqlite3
conn = sqlite3.connect('ishop.db')
curs = conn.cursor()
curs.execute('SELECT * FROM computers ORDER BY count DESC')
rows = curs.fetchall()
print(rows)
curs.close()
conn.close()
```

Отримаємо упорядкування за спаданням по полю кількість:

[(2, ‘Notebook’, 8, 11430.3), (1, ‘PC’, 5, 7570.5), (3, ‘TabletPC’, 4, 3970.2), (4, ‘Console’, 2, 16780.9)]

Зробимо вибірку товарів, які коштують найдорожче:

```
import sqlite3
conn = sqlite3.connect(‘ishop.db’)
curs = conn.cursor()
curs.execute(‘SELECT * FROM computers WHERE price = (SELECT MAX(price) FROM computers)’)
rows = curs.fetchall()
print(rows)
curs.close()
conn.close()
```

Отримаємо результат:

[(4, ‘Console’, 2, 16780.9)]

У зазначеному прикладі оновимо дані у базі даних ishop.db, змінимо кількість TabletPC з 4 на 7:

```
import sqlite3
conn = sqlite3.connect(‘ishop.db’)
curs = conn.cursor()
curs.execute(‘UPDATE computers SET count = 7 WHERE id = 3’)
conn.commit()
curs.execute(‘SELECT * FROM computers WHERE id = 3’)
row = curs.fetchone()
print(row)
curs.close()
conn.close()
```

Таким чином, запис у базі даних із значенням id = 3 зміниться на такий:

(3, ‘TabletPC’, 7, 3970.2)

Виконаємо операцію видалення товарів із бази даних за назвою PC:

```
import sqlite3
conn = sqlite3.connect('ishop.db')
curs = conn.cursor()
curs.execute("DELETE FROM computers WHERE name = 'PC'")
conn.commit()
curs.execute("SELECT * FROM computers")
row = curs.fetchall()
print(row)
curs.close()
conn.close()
```

Отже, товар із назвою PC у базі даних відсутній:

[(2, 'Notebook', 8, 11430.3), (3, 'TabletPC', 7, 3970.2), (4, 'Console', 2, 16780.9)]

Особливості роботи із датою та часом у мові програмування Python

Слід зазначити, що процес роботи з датами та часом викликає багато проблем, оскільки дати можуть бути подані різними способами, наприклад:

July 18 1955
18 Jul 1955
18/7/1955
7/18/1955

Системні значення цієї дати можуть бути неоднозначними, деякі з цих варіантів досить легко визначити, наприклад, 7 – це місяць, а 18 – день місяця, бо місяців всього 12, отже, не буває 18 місяця. Проблема виникає під час читання дати такого вигляду: 12/5/1980.

Крім того, високосний рік – це ще одна проблема. Кожний четвертий рік є високосним, тобто кожний сотий рік не є високосним, а кожен 400-й – є:

```
import calendar
print(calendar.isleap(1955))
print(calendar.isleap(1980))
print(calendar.isleap(2006))
False
True
False
```

Слід зазначити, що робота з часом також може завдати неприємностей, особливо із-за годинних поясів та переходу на літній час.

Стандартна бібліотека мови програмування Python має декілька модулів для роботи з датою та часом, наприклад:

– datetime;

– time;

– calendar.

Модуль datetime у мові програмування Python

У модулі datetime визначено такі об’єкти:

– date – для років, місяців та днів;

– time – для годин, хвилин, секунд та часток секунди;

– datetime – для дати та часу одночасно;

– timedelta – для інтервалів дати та/або часу.

Існує можливість створити об’єкт date, вказавши рік, місяць та день:

```
import datetime
from datetime import date
independence_day = date(1980, 5, 12)
print(independence_day)
datetime.date(1980, 5, 12)
print(independence_day.day)
print(independence_day.month)
print(independence_day.year)
print(independence_day.isoformat())
```

Значення року, місяць та дня будуть доступні як атрибути:

```
1980-05-12
12
5
1980
1980-05-12
```

Слід зазначити, що останній рядок у наведеному коді виводить вміст об’єкту date з використанням функції isoformat() – ISO 8601 (міжнародний стандарт для подання дати та часу). У даному форматі дата записується, починаючи із загального елемента (року) та закінчуючи найточнішим (днем).

Використовуючи такий стандарт можна коректно упорядкувати дати: спочатку по року, а далі по місяцю та по дню.

Зазвичай, формат ISO 8601 вибирають для подання даних у програмах та для назв файлів, які зберігають дані за датою.

Отже, згенеруємо поточну дату, використовуючи функцію today():

```
from datetime import date
now = date.today()
print(now)
```

Отримаємо поточну дату, наприклад:

2018-05-05

Використовуючи об’єкт timedelta, додамо до об’єкта date певний часовий інтервал:

```
from datetime import date
from datetime import timedelta
now = date.today()
one_day = timedelta(days=1)
tomorrow = now + one_day
print(tomorrow)
tomorrow = now + 17*one_day
print(tomorrow)
yesterday = now - one_day
print(yesterday)
```

Отримаємо часовий інтервал, наприклад:

2018-05-06
2018-05-22
2018-05-04

Крім того, об’єкт datetime має функцію now():

```
from datetime import datetime
now = datetime.now()
print(now) – дата та час повністю.
print(now.month) – місяць.
print(now.day) – день.
print(now.hour) – години.
print(now.minute) – хвилини.
print(now.second) – секунди.
print(now.microsecond) – частки секунди.
```

Отримаємо такий результат:

```
2018-05-05 10:11:11.374754
5
5
10
11
11
374754
```

Модуль time у мові програмування Python

Одним із способів подання абсолютного часу є підрахунок кількості секунд, що пройшли з деякої стартової точки.

Функція time() модуля time повертає поточний час як значення epoch (найпростіший спосіб обмінюватися датою та часом між різними системами, використовується кількість секунд, починаючи з 00.00.00 1 січня 1970 року), а функція ctime() конвертує значення epoch у рядок:

```
import time
now = time.time()
print(now)
print(time.ctime(now))
```

Із результату видно, скільки пройшло секунд після настання 1970 року:

```
1525504777.492082
Sat May 5 10:19:37 2018
```

Важливо: значення epoch корисні для обміну датою та часом між різними системами.

Функцію time() модуля time можна використовувати для визначення періодів часу виконання блоків коду програми, для прикладу створимо скрипт, що виводить числа від 20 до 5 включно та повідомлення про час роботи коду:

```
import time
startTime = time.time()
for i in range(20, 4, -1):
    print(i)
endTime = time.time()
print('The program has been running:', str(endTime - startTime), ' seconds.')
```

Отримаємо такий результат:

```
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
The program has been running: 0.1399998664855957 seconds.
```

Крім того, іноді необхідно отримати значення днів або годин. Із модуля time такі значення можна отримати як об’єкти struct_time:

```
import time
now = time.time()
print(time.localtime(now))
print(time.gmtime(now))
```

У даному прикладі використані:
– функція localtime() – надає час у поточному годинному поясі;
– функція gmtime() – надає час у форматі UTC.
Результат використання зазначених функцій такий:

time.struct_time(tm_year=2018, tm_mon=5, tm_mday=5, tm_hour=10, tm_min=30, tm_sec=55, tm_wday=5, tm_yday=125, tm_isdst=1)
time.struct_time(tm_year=2018, tm_mon=5, tm_mday=5, tm_hour=7, tm_min=30, tm_sec=55, tm_wday=5, tm_yday=125, tm_isdst=0)

Слід зазначити, що дату та час можна перетворювати за допомогою функції strftime() із модуля time, яка у виведенні дати та часу використовує специфікатори виведення, подані у таблиці 3.3 [1].

Таблиця 3.3 – Специфікатори виведення для функціїї strftime()

Специфікатор	Одиниця дати/часу	Діапазон
%Y	Рік	1900-… .
%m	Місяць	01-12
%B	Назва місяця	Січень, … .
%d	День місяця	01-31
%A	Назва дня	Неділя, … .
%H	Години (24 години)	00-23
%M	Хвилини	00-59
%S	Секунди	00-59
%f	Мікросекунди	000000, 000001, ..., 999999

Розглянемо приклад роботи функції strftime(), наданої модулем time, вона перетворює об’єкт struct_time у рядок. Визначимо рядок формату під назвою row_format, використавши специфікатори формату з таблиці, та застосуємо функцію localtime() із модуля time:

```
import time
row_format = “It’s %A, %B %d, %Y, local time %H:%M:%S”
t = time.localtime()
print(t)
print(time.strftime(row_format, t))
```

Результат роботи функції strftime() є:

time.struct_time(tm_year=2018, tm_mon=5, tm_mday=5, tm_hour=10, tm_min=41, tm_sec=4, tm_wday=5, tm_yday=125, tm_isdst=1)

It’s Saturday, May 05, 2018, local time 10:41:04

Якщо спробувати аналогічним чином описати об’єкт date, то функція відпрацює тільки дату, а час буде встановлено в 12.00.00:

```
from datetime import date
some_day = date(2006, 2, 21)
row_format = “It’s %B %d, %Y, local time %I:%M:%S%p”
print(some_day.strftime(row_format))
```

Отримаємо такий результат:

It’s February 21, 2006, local time 12:00:00AM

Якщо спробувати аналогічним чином описати об’єкт time, то будуть перетворені тільки частини, що стосуються часу:

```
from datetime import time
row_format = “It’s %B %d, %Y, local time %I:%M:%S%p”
some_time = time(16, 40)
print(some_time.strftime(row_format))
```

Отримаємо такий результат:

It’s January 01, 1900, local time 04:20:00PM

Слід зазначити, що існує і інший шлях для перетворення рядка на дату або час. Для цього використовують функцію strptime() із рядком формату row_format. Рядок із датою та часом повинен точно збігатися із частинами рядка формату. Зазначимо формат «рік-місяць-день», наприклад, 2006-02-21:

```
import time
row_format = “%Y-%m-%d”
print(time.strptime(“2006-02-21”, row_format))
print(time.strptime(“2006-02-21”, row_format)[6]) – понеділок це «0».
print(time.strptime(“2006-02-21”, row_format)[7]) – номер дня у році.
```

Отримаємо такий результат:

time.struct_time(tm_year=2006, tm_mon=2, tm_mday=21, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=1, tm_yday=52, tm_isdst=-1)
1
52

Напишемо скрипт, що повідомляє дату народження для міжнародних друзів із Facebook. На екран буде виведено дату дня народження (місяць, число та день тижня) різними мовами.

Слід зазначити, що модуль locale завантажує формати даних, специфічні для певного культурного оточення.

Щоб визначити, яка локаль за замовчуванням, виконайте такий код:

```
import locale
print(locale.getdefaultlocale())
```

Результатом роботи попереднього фрагменту коду буде кортеж, наприклад, із такими значеннями:

(‘uk_UA’, ‘cp1251’)

Слід зазначити, що gерший рядок кортежу – назва локалі (‘uk_UA’), другий рядок – кодування (‘cp1251’).

Важливо: назва локалі складається із двохсимвольного коду мови, знака підкреслення та двохсимвольного коду країни.

Щоб вивести на екран назви місяців та днів іншими мовами, необхідно змінити свою локаль за допомогою функції setlocale():

– перший аргумент має дорівнювати locale.LC_ALL для дати та часу;

– другий аргумент – це рядок, що містить скорочення мови та країни.

Враховуючи те, що у роботі з локалями та кодуванням можуть виникати помилки, у коді передбачимо це, використовуючи обробники винятків:

```
import locale, datetime
happy = datetime.date(1980, 5, 12)
print(happy.strftime(‘%A, %B %d’))
for lang_country in [‘Belarusian_Belarus.1251’, ‘French_France.1252’, ‘Icelandic_Iceland.1252’, ‘Dutch_Netherlands.1252’, ‘Ukrainian_Ukraine.1251’, ‘Polish_Poland.1250’, ‘pl.UTF-8’, ‘Maori.1252’, ‘Latvian_Latvia.1257’]:
    try:
        locale.setlocale(locale.LC_ALL, lang_country)
        print(happy.strftime(‘%A, %B %d’))
    except locale.Error:
        print(‘unsupported locale setting’)
    except UnicodeEncodeError:
        print(‘can\’t encode characters’)
```

Отримаємо такий результат:

```
Monday, May 12
панядзелак, Май 12
lundi, mai 12
mánudagur, maí 12
maandag, mei 12
понеділок, Травень 12
poniedziałek, maj 12
unsupported locale setting
unsupported locale setting
pirmdiena, maijs 12
```

Крім того, значення усіх lang_country можна дізнатися, використовуючи такий код:

```
names = locale.locale_alias.keys()
print(names)
```

Отримаємо імена тільки тих локалей, які будуть працювати з функцією setlocale(), доповнивши попередній код:

```
good_names = [name for name in names if len(name) == 5 and name[2] == '_']
print(good_names)
```

Слід зазначити, якщо необхідно отримати значення локалей для конкретної країни, наприклад, Франції, то доповнимо попередній код рядками:

```
fr = [name for name in good_names if name.startswith('fr')]
print(fr)
```

Отримаємо такий код:

```
import locale, datetime
names = locale.locale_alias.keys()
good_names = [name for name in names if len(name) == 5 and name[2] == '_']
fr = [name for name in good_names if name.startswith('fr')]
print(fr)
```

Список локалей для Франції виявиться таким:

```
['fr_ch', 'fr_be', 'fr_ca', 'fr_fr', 'fr_lu']
```

СПИСОК РЕКОМЕНДОВАНИХ ДЖЕРЕЛ

1. Путівник мовою програмування Python [Електронний ресурс]. – Режим доступу : http://pythonguide.rozh2sch.org.ua. – (дата звернення : 22.01.2018). – Назва з екрану.

2. ArcGIS Pro Python [Електронний ресурс]. – Режим доступу : https://pro.arcgis.com/ru/pro-app/arcpy/main/arcgis-pro-arcpy-reference.htm. – (дата звернення : 23.01.2018). – Назва з екрану.

3. Python. Обучение программированию [Электронный ресурс]. – Режим доступа : https://younglinux.info/python. – (дата обращения : 24.01.2018). – Загл. с экрана.

4. Програмування на мові Python (3.x). Початковий курс [Електронний ресурс]. – Режим доступу : https://sites.google.com/site/pythonukr/urok-15-perevirocna-robota-z-osnov-programuvanna-na-python. – (дата звернення : 25.01.2018). – Назва з екрану.

5. Лутц М. Изучаем Python / М. Лутц. – СПб. : Символ-Плюс, 2011. – 1280 с.

6. Лутц М. Программирование на Python : в 2 томах / М. Лутц. – СПб. : Символ-Плюс, 2011. – Т. 1. – 992 с.

7. Дэвид М. Бизли Python. Подробный справочник / Дэвид М. Бизли. – СПб. : Символ-Плюс, 2010. – 864 с.

8. Саммерфилд М. Программирование на Python 3. Подробное руководство / М. Саммерфилд. – СПб. : Символ-Плюс, 2009. – 608 с.

9. Саммерфилд М. Python на практике / М. Саммерфилд – М. : ДМК Пресс, 2014. – 338 с.

10. Сузи Р. А. Язык программирования Python : учеб. пособие / Р. А. Сузи. – М. : ИНТУИТ, БИНОМ. Лаборатория знаний, 2006. – 328 с.

11. Доусон М. Программируем на Python / М. Доусон. – СПб. : Питер, 2012. – 432 с.

12. Хахаев И. А. Практикум по алгоритмизации и программированию на Python: учебник / И. А. Хахаев. – М. : Альт Линукс, 2010. – 126 с.

13. Дэвид М. Бизли Язык программирования Python. Справочник / Дэвид М. Бизли. – Киев : ДиаСофт, 2000. – 336 с.

14. Сузи Р. А. Python. Наиболее полное руководство / Р. А. Сузи. – СПб. : БХВ-Петербург, 2002. – 768 с.

15. Шипулін В. Д. Основні принципи геоінформаційних систем / В. Д. Шипулін. – Харків : ХНАМГ, 2012. – 312 с.

Навчальне видання

ТВОРОШЕНКО Ірина Сергіївна

СПЕЦІАЛІЗОВАНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

КОНСПЕКТ ЛЕКЦІЙ

(для магістрів денної та заочної форм навчання спеціальності 193 – Геодезія та землеустрій освітньої програми «Геодезія та землеустрій»)

Відповідальний за випуск *К. А. Мамонов*

За авторською редакцією

Комп'ютерний набір *І. С. Творошенко*

Комп'ютерне верстання *І. В. Волосожарова*

План 2018, поз. 37 Л

Підп. до друку 13.02.2018. Формат 60 × 84/16.
Друк на ризографі. Ум. друк. арк. 4,3.
Тираж 50 пр. Зам. № .

Видавець і виготовлювач:
Харківський національний університет
міського господарства імені О. М. Бекетова,
вул. Маршала Бажанова, 17, Харків, 61002.
Електронна адреса: rectorat@kname.edu.ua.
Свідоцтво суб'єкта видавничої справи:
ДК № 5328 від 11.04.2017.