МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ
Факультет інформаційних технологій

Д.Г. МЕДВЕДЄВ

НАВЧАЛЬНО-МЕТОДИЧНИЙ ПОСІБНИК

з дисципліни «Основи програмування на Python»

КРИВИЙ РІГ 2023

Рецензенти:

В.Б. Хоцкіна, к.т.н., доцент кафедри інформатики та прикладного програмного забезпечення, Державний університет економіки і технологій

П.В. Мерзликін, к.ф-м.н., доцент кафедри інформатики та прикладної математики Криворізького державного педагогічного університету

Рекомендовано до друку навчально-методичною радою Державного університету економіки і технологій протокол №10 від 27.03.2023.

Д.Г. Медведєв: Навчально-методичний посібник з освітньої компоненти «Основи програмування на Python» для здобувачів вищої освіти освітньо-кваліфікаційного рівня «бакалавр» спеціальності 121 «Інженерія програмного забезпечення» денної та заочної форм навчання. – Кривий Ріг: Державний університет, економіки і технологій, 2023. –84 с.

Подано основи програмування мовою Python. Розглянуто принципи роботи інтерпретатора; типи даних (числа, змінні та незмінні структури даних, такі як рядки, кортежі, списки, словники, множини, оператори та методи роботи з ними); оператор вибору варіанта дії на основі результатів перевірки (оператор if-else, оператор try-catch), циклічні конструкції (оператори while та for) та процес створення функцій.

Навчально-методичний посібник розроблено у відповідності до навчального плану з метою надання здобувачам вищої освіти допомоги у освоєнні теоретичного матеріалу та при виконанні ними лабораторних та індивідуальних завдань. Посібник призначений для здобувачів вищої освіти освітньо-кваліфікаційного рівня «бакалавр» спеціальності 121 «Інженерія програмного забезпечення» денної та заочної форм навчання.

Зміст

Розділ 1. Початок роботи

Python є програмний пакет, званий інтерпретатором. Інтерпретатор – це різновид програми, яка виконує інші програми. Коли ви пишете програму на Python, інтерпретатор Python читає її і виконує інструкції, що містяться в ній. Фактично інтерпретатор є рівнем програмної логіки між вашим кодом та обладнанням комп'ютера.

В результаті встановлення Python на машині створюється кілька компонентів – мінімум інтерпретатор та бібліотека підтримки. Залежно від способу застосування інтерпретатор Python може набувати форми програми, що виконується, або набору бібліотек, пов'язаних з іншою програмою. Залежно від використовуваного різновиду Python, сам інтерпретатор може бути реалізований у вигляді програми C, набору класів Java або чогось іншого. Незалежно від форми, код Python, який ви пишете, завжди повинен виконуватися інтерпретатором. Щоб зробити це можливим, ви повинні встановити на комп'ютері інтерпретатор Python.

- Користувачі Windows завантажують і запускають самовстановлюваний виконуваний файл, який поміщає Python на їхні комп'ютери. Потрібно просто двічі клацнути на імені файлу і ствердно відповідати на наступні запити.
- Користувачі Linux і Mac OS X, можливо, вже мають заздалегідь встановлену копію Python на своїх комп'ютерах - в даний час це стандартний компонент для їх платформ.
- Деякі користувачі Linux і Mac OS X (і більшість користувачів Unix) компілюють Python з пакета з повним вихідним кодом.
- Користувачі Linux також можуть знайти файли RPM, а користувачі Mac OS X — різноманітні інсталяційні пакети, специфічні для Mac.
- На інших платформах передбачені прийоми установки. Наприклад, Python доступний на мобільних телефонах, планшетах, ігрових

приставках та пристроях iPod, але деталі установки варіюються в широких межах.

Сам Python можна завантажити з основного веб-сайту http://www.python.org. Він також доступний через багато інших каналів розповсюдження. Не забувайте перед встановленням перевіряти, чи є вже копія Python у системі.

У своїй простій формі програма на Python є лише текстовим файлом, що містить оператори Python. Наприклад, у наведеному нижче вмісті файлу на ім'я script0 .py знаходиться найпростіший сценарій Python, який є повнофункціональною програмою Python:

```
print('hello world')
print(2 ** 100)
```

Файл scriptO.py містить два Python-оператора print, які виводять у вихідний потік рядок (текст у лапках) та результат числового виразу (2 у ступені 100). Поки що не турбуйтеся про синтаксис наведеного коду — зараз цікавить лише приведення його в готовність до запуску.

Ви можете створити такий файл із операторами у будь-якому текстовому редакторі, який вам подобається. За згодою файлів з програмами Python призначаються імена, що закінчуються на . ру; формально така схема іменування є обов'язковою лише для файлів, які “імпортуються”, але задля узгодженості більшість файлів Python мають імена . ру.

Після набору всіх операторів у текстовому файлі ви повинні повідомити інтерпретатор Python про необхідність виконати файл — що просто означає запуск всіх операторів у файлі спочатку до кінця, один за одним. Як пояснюється в наступному розділі, ви можете запускати файли програм Python через рядки командної оболонки, клацнувши на їх значках, з IDE-середовищ та за допомогою інших стандартних прийомів. Якщо виконання файлу проходить гладко, тоді ви побачите результати двох операторів print у якомусь місці — за замовчуванням у тому вікні, де запускалася програма:

```
hello world
1267650600228229401496703205376
```

Після запуску програми Python внутрішньо і майже повністю приховано від вас спочатку компілює ваш вихідний код (оператори всередині файлу) у формат відомий як байт-код. Компіляція є просто кроком трансляції, а байт-код це низькорівневе і незалежне від платформи уявлення вихідного коду. Грубо кажучи, Python транслює кожен оператор вихідного коду групу інструкцій байт-коду, розбиваючи їх у окремі кроки. Така трансляція в байт-код відбувається зі швидкістю виконання - байт-код здатний виконуватися швидше, ніж початкові оператори вихідного коду текстового файлу.

Ви, напевно, помітили, що в попередньому абзаці було вказано, що компіляція майже повністю прихована від вас. Якщо процес Python має доступ до запису на вашому комп'ютері, тоді він зберігатиме програми у файлах з розширенням .pyc (означає скомпільований (compiled) вихідний файл .py). До виходу версії Python 3.2 ці файли з'являлися комп'ютері після запуску кількох програм там, де були відповідні файли вихідного коду, тобто. у тих самих каталогах. Наприклад, після імпортування script .py з'являвся файл script .pyc.

У Python 3.2 та наступних версіях файли байт-коду .pyc натомість зберігаються в підкаталозі на ім'я__ pycache__ , розташованому в каталозі, де знаходяться файли вихідного коду, та їх імена ідентифікують версію Python, в якій вони створювалися (скажімо, script.cpython-3 pyc). Новий підкаталог__ pycache__ допомагає уникнути безладу, а нова угода про іменування файлів байт-коду запобігає переписуванню збереженого байт-коду різними версіями Python, які можуть бути встановлені на одному комп'ютері. В обох моделях Python зберігає байт-код подібного роду як оптимізацію швидкості початкового завантаження. Під час наступного запуску програми Python завантажить файли .pyc і пропустить крок компіляції за умови, що ви не змінювали вихідний код після збереження байт-коду і не виконуєте версію Python, яка відрізняється від тієї, яка створювала байт-код.

В результаті внесення змін до вихідного коду та застосування різних номерів версій Python ініціюватиме створення нового файлу байт-коду. Якщо Python не може записувати файли байт-коду на вашому комп'ютері, то

програма все одно буде працювати — байт-код генерується в пам'яті і просто відкидається після завершення виконання програми. Тим не менш, оскільки файли .pyc прискорюють завантаження, для великих програм є сенс забезпечити можливість записування файлів байт-коду. Файли байт-коду є також одним із способів постачання програм Python - вони будуть виконуватися, навіть якщо все, що вдається знайти - це файли .pyc, а файли вихідного коду .py відсутні.

Нарешті, майте на увазі, що байт-код зберігається лише для тих файлів, що імпортуються, але не для файлів верхнього рівня програми, що виконуються лише як сценарії (точно, йдеться про оптимізацію імпорту).

Після того, як програма скомпільована в байт-код (або байт-код був завантажений з існуючих файлів .pyc), вона відправляється на виконання того, що в загальному випадку відоме як віртуальна машина Python (Python Virtual Machine - PVM). Машина PVM справляє глибше враження, ніж є насправді; насправді вона не є окремою програмою і сама по собі не потребує встановлення. По суті, машина PVM — лише великий закодований цикл, який проходить за інструкціями вашого байт-коду, один за одним, щоб виконати їх операції. Машина PVM - це виконуючий механізм Python; вона завжди присутня у вигляді частини системи Python і є компонентом, який по-справжньому виконує ваші сценарії. Формально вона є останнім кроком того, що називається “інтерпретатор Python”.

Альтернативні реалізації Python

Зараз є принаймні п'ять реалізацій мови Python — CPython, Jython, IronPython, Stackless і PyPy. Незважаючи на те, що між ними є багато взаємно доповнюючих ідей і робіт, кожна реалізація є програмною системою, що окремо встановлюється, з власними розробниками і користувальницькою базою. До інших потенційних кандидатів входять системи Cython і Shed Skin - інструменти оптимізації, тому що не реалізують стандартну мову Python (перша є сумішшю Python/C, а друга — неявно статично типізованою).

Коротко кажучи, CPython — це стандартна реалізація, яку забажають використати більшість читачів (якщо ви не впевнені, то навряд чи будете винятком). Всі інші реалізації Python призначені для специфічних цілей та ролей, але вони часто можуть виконувати також більшість ролей CPython. Всі вони реалізують ту саму мову Python, але виконують програми різними способами.

Наприклад, PyPy є заміною CPython, яка дозволяє виконувати багато програм набагато швидше. Подібним чином Jython і IronPython являють собою абсолютно незалежні реалізації Python, які компілюють вихідний код Python для різних архітектур часу виконання, щоб забезпечити прямий доступ до компонентів Java і .NET. Доступ до програмного забезпечення Java і .NET також можливий із стандартних програм CPython — наприклад, JPypeia Python for .NET дозволяють стандартному коду CPython звертатися до компонентів Java і .NET. Системи Jython та IronPython пропонують більш завершені рішення, надаючи повні реалізації мови Python.

Розділ 2 Типи даних

Якщо ви мали справу з мовою нижчого рівня на кшталт C або C++, то знаєте, що більшість роботи сконцентрована на реалізації обзехшо#, також відомих, як структури даних, для представлення компонентів у вашій предметній області. Вам потрібно планувати структури в пам'яті, керувати виділенням пам'яті, створювати процедури пошуку та доступу тощо. Такі рутинні завдання утомливі (і загрожують помилками) і вони найчастіше відволікають від справжніх цілей програми.

У типових програм більшість рутинних робіт зникає. Через те, що Python надає потужні типи об'єктів як невід'ємну частину мови, зазвичай немає необхідності написання коду реалізації об'єктів перед початком вирішення завдань. По суті, якщо вам не потрібно спеціальна обробка, яку вбудовані типи не забезпечують, тоді майже завжди краще застосовувати вбудований об'єкт, а не реалізовувати власний. Нижче наведено причини.

• Вбудовані об'єкти полегшують написання програм. При вирішенні найпростіших завдань вбудовані типи часто є необхідним уявлення структури предметної області. Оскільки ви безкоштовно отримуєте потужні інструменти на кшталт колекцій (списків) та пошукових таблиць (словників), то можете використовувати їх негайно. За допомогою винятково вбудованих типів об'єктів Python ви можете виконати велику роботу.

• Вбудовані об'єкти є компонентами розширень. Для більш складних завдань може знадобитися надати власні об'єкти із застосуванням класів Python або інтерфейсів мови C. Реалізовані вручну об'єкти нерідко будуються на основі типів, подібних до списків і словників. Наприклад, структура даних стека може бути реалізована у вигляді класу, який керує або налаштовує вбудований список.

• Вбудовані об'єкти часто ефективніші за спеціальні структури даних. Вбудовані типи Python використовують вже оптимізовані алгоритми для роботи зі структурами даних, які реалізовані на C, щоб забезпечувати високу

швидкодію. Незважаючи на можливість самостійного написання схожих типів об'єктів, вам зазвичай буде важко досягти рівня продуктивності, що пропонується вбудованими типами об'єктів.

• Вбудовані об'єкти є стандартною частиною мови. У певному сенсі Python багато чого запозичує з мов, які спираються на вбудовані інструменти (скажімо, LISP), і мов, які покладаються на те, що реалізації інструментів або фреймворків надасть програміст (наприклад, C++). Хоча ви можете реалізувати унікальні типи об'єктів Python, вам не доведеться робити це для того, щоб розпочати роботу: Більше того, оскільки вбудовані типи об'єктів Python є стандартом, вони завжди такі самі; з іншого боку, патентовані фреймворки мають тенденцію відрізнятися від майданчика до майданчика.

Іншими словами, вбудовані типи об'єктів не тільки полегшують програмування, але також є більш потужними та ефективними, ніж більшість того, що можна створити з нуля. Незалежно від того, чи ви реалізуєте нові типи об'єктів, вбудовані типи об'єктів утворюють основу кожної програми Python.

Основні вбудовані типи об'єктів:

- Числа – 1234, 3.1415, 3+4j, 0ЫІ11, Decimal (), Fraction ()
- Рядки – 'spam’, "Bob' s", b’a\x01c’, u’sp\xc4m’
- Списки – [1, [2, ’three’], 4.5], list(range (10))
- Словники – {’food’: 'spam', ’taste': 'yum’}, dict(hours=10)
- Кортежі – (1, 'spam', 4, ’U’), tuple('spam’), namedtuple
- Файли – open('eggs.txt'), open(r’C:\ham.bin', ' wb')
- Множини – set('abc'), {'a', 'b', ’c'}
- Інші основні типи – булеві значення, None
- Типи програмних одиниць –функції, модулі, класи
- Типи, пов'язані з реалізацією – скомпільований код, трасування стека

2.1. Числа

Набір основних об'єктів Python включає очікувані типи: цілі числа, що не мають дробової частини, числа з плаваючою точкою, які мають дробову частину, і більш екзотичні типи - комплексні числа з уявною частиною, десяткові числа з фіксованою точністю, раціональні числа з чисельником і знаменником, а також повнофункціональні множини. Вбудованих чисел цілком достатньо для представлення більшості числових величин (від вашого віку до сальдо вашого банківського рахунку), але є ще більше типів у вигляді сторонніх доповнень.

Незважаючи на пропозицію ряду химерних варіантів, основні числові типи Python, загалом, є базовими. Числа Python підтримують звичайні математичні операції. Наприклад, плюс (+) виконує додавання, зірочка (*) використовується для множення, а дві зірочки (**) застосовуються для зведення в ступінь:

```
>>> 123 + 222
345
>>> 1.5 * 4
6.0
>>> 2 ** 100
1267650600228229401496703205376
```

Зверніть увагу на результат останньої операції: цілочисленний тип Python 3.X при необхідності автоматично забезпечує підвищену точність для великих чисел (у Python 2.X числа, надто великі для звичайного цілочисленного типу, підтримувалися окремим довгим цілочисленним типом). Скажімо, в Python ви можете обчислити 2 у ступеню 1 000 000 у вигляді цілого числа, але ймовірно не повинні виводити результат - він міститиме понад 300 000 цифр, тому доведеться почекати:

```
>>> len(str(2 ** 1000000))
301030
```

Така форма вкладених дзвінків працює зсередини назовні - спочатку результуюче число операції * * перетворюється на рядок цифр за допомогою вбудованої функції str, після чого за допомогою 1еп виходить довжина

підсумкового рядка. Кінцевим результатом буде кількість цифр. Функції str і 1еп працюють з багатьма типами об'єктів.

У версіях, що передують Python 2.7 і Python 3.1, після початку експериментування з числами з плаваючою точкою ви ймовірно зустрінете дещо, що на перший погляд здається дивним:

```
>>> 3.1415 * 2
6.2830000000000004
>>> print(3.1415 * 2)
6.283
```

Перший результат не є помилкою; це проблема відображення. Насправді є два способи виведення будь-якого об'єкта в Python – з повною точністю (як у першому результаті) та у формі, дружній до користувача (як у другому результаті). Формально перша форма називається герг (вид об'єкта як у коді), а друга - str (вигляд, дружній до користувача). У старих версіях Python форма герг для чисел з плаваючою точкою часом відображала з більшою точністю, ніж очікувалося. Різниця також може бути значущою, коли ми підійдемо до використання класів

Крім виразів до складу Python входить кілька корисних числових модулів (модулі - це пакети додаткових інструментів, які ми імпортуємо для їх застосування):

```
>>> import math
>>> ma th. pi
3.141592653589793
>>> math. sqrt(85)
9.219544457292887
```

Модуль math містить більш складні числові інструменти у вигляді функцій, а модуль random виконує генерацію випадкових чисел і випадковий вибір (тут із списку Python, що задається в квадратних дужках, — упорядкованої колекції інших об'єктів):

```
>>> import random
>>> random. random ()
0.7082048489415967
```

```
>>> random. choice ([1, 2, 3, 4])
1
```

Python також включає більш екзотичні числові об'єкти, такі як комплексні числа, числа з фіксованою точністю, раціональні числа, множини і булевські числа, а в області сторонніх розширень з відкритим кодом є ще більше числових об'єктів (наприклад, матриці, вектори та числа з підвищеною точністю).

2.2. Рядки

Рядки застосовуються для запису текстової інформації (скажімо, вашого імені) та довільних сукупностей байтів (на кшталт вмісту файлу зображення). Вони є першим прикладом того, що Python називається послідовністю — позиційно впорядкованою колекцією інших об'єктів. Для елементів послідовності, що містяться, підтримують порядок зліва направо: елементи зберігаються і витягуються за своїми відносними позиціями. Строго кажучи, рядки є послідовністю односимвольних рядків.

Як послідовності рядки підтримують операції, які передбачають наявність позиційного порядку серед елементів. Наприклад, якщо ми маємо чотирисимвольний рядок, записаний у лапках (зазвичай одинарних), то можемо перевірити її довжину за допомогою вбудованої функції 1еп та витягти її компоненти за допомогою виразів індексації:

```
>>> S = 'Spam'
>>> len(S)
4
>>> S [0]
'S'
>>> S[1]
'p'
```

Індекси в Python є зсувом спереду і тому починаються з 0: перший елемент знаходиться за індексом 0, другий - за індексом 1 і т.д.

Зверніть увагу, як тут рядок присвоюється змінною на ім'я S. Деталями роботи присвоювання ми займемося пізніше (головним чином розділ 6), але змінні Python ніколи не потрібно оголошувати заздалегідь. Змінна створюється, коли ви надаєте їй значення, яке може бути об'єктом будь-якого типу, і вона замінюється наданим значенням, коли з'являється у виразі. Змінною має бути щось присвоєно на час використання її значення. Для збереження об'єкта з метою подальшої роботи з ним необхідно присвоїти його змінну.

У Python ми можемо також індексувати у зворотному напрямку, з кінця, тобто. позитивні індекси вважаються зліва, а негативні - праворуч:

```
>>> S [ —1]
‘m’
>>> S [—2]
’a’
```

Формально негативний індекс додається до довжини рядка, тому наступні дві операції еквівалентні (хоча першу простіше записувати і менш легко припуститися помилки):

```
>>> S[-1]
'm'
>>> S[len(S)-1]
’m’
```

Зверніть увагу, що у квадратних дужках ми можемо застосовувати довільний вираз, а не тільки жорстко закодований числовий літерал - скрізь, де Python очікує значення, допускається використовувати літерал, змінну або будь-який бажаний вираз. У цьому плані синтаксис Python повністю універсальний.

На додаток до простої позиційної індексації послідовності також підтримують більш загальну форму індексації, відому як нарізування, яке є способом вилучення цілої частини (зрізу) за один крок. Ось приклад:

```
>>> S
’Spam'
>>> S [1:3]
```

```
'pa'
```

Ймовірно, найпростіше думати про зрізи як про спосіб вилучення з рядка цілого розділу за один крок. Універсальна форма зрізу, X [I: J], означає "надати з X весь вміст, починаючи зі зміщення I і закінчуючи зсувом J, не включаючи його". Результат повертається до нового об'єкта. Друга з наведених вище операцій видає всі символи рядка S зі зміщення 1 до 2 (тобто зі зміщення 1 до 3-1) у вигляді нового рядка. Наслідком виявляється нарізання або "вибірка" двох символів із середини.

За замовчуванням ліва межа зрізу приймається рівною нулю, а права — довжині послідовності, що нарізається. Це призводить до кількох поширених варіантів застосування:

```
>>> S[1:]
'pam'
>>> s
'Spam'
>>> S[0:3]
'Spa'
>>> S[:3]
'Spa'
>>> S[:-1]
'Spa'
>>> S[:]
'Spam'
```

У другій з кінця операції демонструється можливість використання негативних зсувів у межах зрізів, а остання операція фактично копіює цілий рядок. Як ви дізнаєтеся пізніше, копіювати рядок немає сенсу, але така форма може бути корисною в послідовностях, подібних до списків.

Нарешті, будучи послідовностями, рядки підтримують також конкатенацію (об'єднання двох рядків у новий рядок), що позначається знаком "плюс", і повторення (створення нового рядка шляхом повторення іншого):

```
>>> s
'Spam'
```

```
>>> s + 'xyz'
'Spamxyz'
>>> S
'Spam'
>>> S * 8
'SpamSpamSpamSpamSpamSpamSpamSpam'
```

Важливо, що знак "плюс" (+) для різних об'єктів означає дії, що відрізняються: додавання для чисел і конкатенацію для рядків. Це загальна властивість Python, яка називається поліморфізмом, — коротко кажучи, сенс операції залежить від об'єктів, до яких вона застосовується. Під час вивчення динамічної типізації ви побачите, що властивість поліморфізму пояснює більшість причин лаконічності та гнучкості коду Python. Оскільки типи не обмежені, операція Python здатна нормально працювати з багатьма різними типами об'єктів автоматично за умови, що вони підтримують сумісний інтерфейс (подібно до операції + тут).

У попередніх прикладах також зверніть увагу, що виконання всіх операцій не викликало зміни вихідного рядка. Кожна рядкова операція визначена так, щоб робити як результат новий рядок, тому що рядки в Python є незмінними - після створення їх не можна модифікувати на місці. Іншими словами, ви ніколи не перезапишете значення незмінних об'єктів. Наприклад, ви не зможете змінити рядок, присвоюючи значення символу в одній з її позицій, але завжди зможете побудувати новий рядок і призначити йому те саме ім'я. Зважаючи на те, що під час роботи Python очищає старі об'єкти, це не настільки неефективно, як може здатися.

Кожен об'єкт у Python класифікується як незмінний (немодифікований) чи ні. Щодо основних типів, то числа, рядки та кортежі незмінні, а списки, словники та множини — ні; вони можуть вільно модифікуватися на місці, як більшість нових об'єктів, які ви будете створювати за допомогою класів. Така відмінність виявляється критично важливою у роботі Python, але ми поки що не в змозі повністю дослідити його вплив. Крім того, незмінність можна використовувати для гарантування того, що об'єкт залишається постійним

протягом усієї програми; значення об'єктів, що змінюються, здатні змінюватися в будь-який момент і в будь-якому місці (очікуєте ви цього чи ні).

```
>>> S = ’shrubbery’
>>> L = list(S)
>>> L
[’s’, ’h’, ’r’, ’u’, ’b’, ’b’, ’e’, ’r’, ’y’]
>>> L[l] = ’c’
>>> ’ ’ .join(L)
’scrubbery'
>>> B = bytearray (b ’ spam ’)
>>> B. extend (b ’ eggs ’)
>>> в
bytearray(b’spameggs’)
>>> B.decode()
’spameggs’
```

Тип bytearray підтримує зміни на місці для тексту, але тільки щодо символів, які мають ширину не більше 8 бітів (скажімо, ASCII). Всі інші рядки, як і раніше, незмінні — bytearray є особливим гібридом незмінних байтових рядків (з синтаксисом b ' . . ' , обов'язковим у Python 3.X і необов'язковим у Python 2.X) та змінюваних списків (позначаються та відображаються)).

Рядки, як об'єкти Python, мають методи (тобто функції, які виконують самі об'єкти):

- S.find(str, [start],[end]) — пошук підрядка в рядку, повертає номер першого входження або -1;
- S.rfind(str, [start],[end]) — пошук підрядка в рядку, повертає номер останнього входження або -1;
- S.index(str, [start],[end]) — пошук підрядка в рядку, повертає номер першого входження або викликає ValueError;
- S.rindex(str, [start],[end]) — пошук підрядка в рядку, повертає номер останнього входження або викликає ValueError;
- S.replace(шаблон, заміна) — заміна шаблону;
- S.split(символ) — розбиття рядка за роздільником;

- S.isdigit() — чи складається рядок з цифр;
- S.isalpha() — чи складається рядок з літер;
- S.isalnum() — чи складається рядок з цифр або літер;
- S.islower() — чи складається рядок із символів у нижньому регістрі;
- S.isupper() — чи складається рядок із символів у верхньому регістрі;
- S.isspace() — чи складається рядок з невідображуваних символів (пробіл, ознаки кінця сторінки '\f' і рядка '\n', переведення каретки '\r', горизонтальна табуляція '\t' і вертикальна табуляція '\v');
- S.istitle() — чи починаються слова в рядку з великої літери;
- S.upper() — перетворення рядка до верхнього регістру;
- S.lower() — перетворення рядка до нижнього регістру;
- S.startswith(str) — чи починається рядок S з шаблону str;
- S.endswith(str) — чи закінчується рядок S шаблоном str;
- S.join(список) — збірка рядка зі списку з роздільником S;
- ord(символ) — ASCII код символу;
- chr(число) — символ з вказаним кодом ASCII;
- S.capitalize() — переводить перший символ рядка у верхній регістр, а всі інші — в нижній;
- S.center(width, [fill]) — повертає відцентрований рядок, по краях якої стоїть символ fill (пробіл за замовчуванням);
- S.count(str, [start],[end]) — повертає кількість непересічних входжень підрядка в діапазоні [початок, кінець] (0 і довжина рядка як усталено);
- S.expandtabs([tabsize]) — повертає копію рядка, в якому всі символи табуляції замінтено одним або кількома пропусками залежно від поточного стовпчика. Якщо TabSize не вказано, розмір табуляції — 8 пробілів;
- S.lstrip([chars]) — видалення символів пробілів на початку рядка;
- S.rstrip([chars]) — видалення символів пробілів в кінці рядка;
- S.strip([chars]) — видалення символів пробілів на початку і в кінці рядка;

- S.partition(шаблон) — повертає кортеж, що містить частину перед першим шаблоном, сам шаблон, і частина після шаблону. Якщо шаблон не знайдено, повертається кортеж, що містить самий рядок, а потім два порожніх рядки;
- S.rpartition(sep) — повертає кортеж, що містить частину перед останнім шаблоном, сам шаблон, і частина після шаблону, якщо шаблон не знайдений, повертається кортеж, що містить два порожні рядки, а потім самий рядок;
- S.swapcase() — перекладає символи нижнього регістра в верхній, а верхнього — в нижній;
- S.title() — першу букву кожного слова переводить в верхній регістр, а всі інші в нижній;
- S.zfill(width) — робить довжину рядка не меншою width, в разі потреби заповнює перші символи нулями;
- S.ljust(width, fillchar=" ") — робить довжину рядка не меншою width, в разі потреби заповнює останні символи символом fillchar;
- S.rjust(width, fillchar=" ") — робить довжину рядки не меншою width, в разі потреби заповнює перші символи символом fillchar;

2.3. Списки

Об'єкт списку Python є найбільш загальною послідовністю, що пропонується мовою. Списки є позиційно впорядкованими колекціями об'єктів довільних типів і не мають фіксованих розмірів. Крім того, вони змінюються - на відміну від рядків списки можна модифікувати на місці шляхом присвоєння зсувів і виклику різноманітних спискових методів. Відповідно вони надають дуже гнучкий інструмент для представлення довільних колекцій – переліку файлів у каталозі, співробітників у компанії, повідомлень у скриньці вхідної пошти тощо.

Будучи послідовностями, списки підтримують усі операції над послідовностями, які ми обговорювали для рядків; єдина відмінність у цьому, що результатами зазвичай будуть рядки, а списки. Наприклад, маючи список із трьох елементів. Ми можемо його індексувати, нарізати та виконувати інші дії в точності, як чинили з рядками:

```
>>> L = [123, ’spam', 1.23]
>>> len(L)
3
>>> L[0]
123
>>> L[:-1]
[123, 'spam']
>>> L + [4, 5, 6]
[123, 'spam', 1.23, 4, 5, 6]
>>> L * 2
[123, 'spam', 1.23, 123, 'spam', 1.23]
>>> L
[123, 'spam', 1.23]
```

Списки Python можуть нагадувати масиви іншими мовами, але вони, як правило, більш потужні. Насамперед, немає обмеження, що елементи повинні належати якомусь фіксованому типу — скажімо, наведений вище список містив три об'єкти абсолютно різних типів (ціле число, рядок і число з плаваючою точкою). Крім того, списки не мають фіксованих розмірів. Тобто вони можуть збільшуватися та зменшуватися у відповідь на операції, специфічні для списків:

```
>>> L.append('N1’)
>>> L
[123, ’spam’, 1.23, 'N1’]
>>> L.pop(2)
1.23
>>> L
[123, 'spam', 'N1' ]
```

Списковий метод append збільшує розмір списку та поміщає об'єкт у кінець; метод pop (або еквівалентний оператор del) видаляє елемент із заданим зміщенням, призводячи до зменшення списку. Інші спискові методи вставляють об'єкт у довільну позицію (insert), видаляють зазначений елемент за значенням (remove), додають безліч елементів до кінця (extend) і т.д. Оскільки списки змінюються, більшість спискових методів також модифікують об'єкт списку на місці, а не створюють новий такий об'єкт:

```
>>> M = ['ЬЬ', 'aa', 'cc']
>>> M.sort ()
>>> м
['aa', 'ЬЬ', 'cc']
>>> M. reverse ()
>>> M
['cc', 'ЬЬ', 'aa']
```

Одна приємна особливість основних типів даних Python полягає в тому, що вони підтримують довільне вкладення – ми можемо вкладати їх у будь-якій комбінації та будь-яку бажану глибину. Скажімо, у нас може бути список, який містить словник, що містить ще один список і т.д. Одне з прямих застосувань такої можливості пов'язане з поданням у Python матриць, або багатовимірних масивів. Список вкладених списків придатний для базових додатків (у рядках 2 і 3 ви отримаєте запрошення . . . під час роботи у деяких інтерфейсах, але з IDLE):

```
>>> M = [ [1, 2, 3] ,
[4, 5, 6],
[7, 8, 9]]
>>> M
[[1 , 3, 3], [4, 5, б], [7, 8, 9]]
```

Тут ми створюємо список, який містить три інші списки. Результатом буде представлення матриці 3x3 чисел. Доступ до такої структури може здійснюватися різноманітними способами:

```
>>> M[1]
[4, 5, б]
```

```
>>> M[1][2]
6
```

Перша операція витягує другий рядок цілком, а друга — третій елемент даного рядка (вона виконується зліва направо, як операції strip і split для рядків, що раніше застосовувалися). Ув'язування разом операцій індексації дозволяє нам переміщатися дедалі глибше у структуру вкладених об'єктів.

2.4. Словники

Словники Python – щось зовсім інше; вони взагалі є послідовностями і замість відомі як відображення. Відображення також є колекціями інших об'єктів, але вони зберігають об'єкти за ключами, а не за відносними позиціями. Насправді відображення не підтримують надійний порядок зліва направо; вони просто відображають ключі на пов'язані значення. Словники - єдиний тип відображення в наборі основних об'єктів Python - є змінними: як і списки, їх можна модифікувати на місці і здатні збільшуватися і зменшуватися на вимогу. Нарешті, подібно до списків словники - це гнучкий інструмент для представлення колекцій, але їх мнемонічні ключі краще підходять, коли елементи колекції іменовані або позначені, скажімо, як поля в записі бази даних.

При написанні у вигляді літералів словники вказуються у фігурних дужках і складаються з пар “ключ: значення”. Словники зручні завжди, коли нам необхідно асоціювати набір значень із ключами — наприклад, для опису властивостей чогось. Розглянемо наступний словник їх трьох елементів (з ключами ' food ', ' quantity ' і ' color ', які можуть представляти деталі позиції гіпотетичного меню):

```
>>> D = {’food’: ’Spam', ’quantity’: 4, ’color’: 'pink'}
```

Ми можемо індексувати цей словник за ключем, щоб отримувати та змінювати значення, пов'язані з ключами. Операція індексації словника має такий самий синтаксис, як для послідовностей, але елементом у квадратних дужках буде ключ, а не відносна позиція:

```
>>> D['food']
'Spam'
>>> D['quantity'] += 1
>>> D
{'color': ’pink’, 'food': 'Spam', 'quantity': 5}
```

Хоча форма літералу у фігурних дужках зустрічається, мабуть, частіше доводиться бачити словники, побудовані іншими способами (всі ці програми рідко відомі до її запуску). Скажімо, наступний код починається з порожнього словника і заповнює його одним ключем за раз. На відміну від присвоєння елементу у списку, що знаходиться поза встановленими межами, яке заборонено, привласнення нового ключа словника призводить до створення цього ключа:

```
>>> D = {}
>>> D['name'] = 'Bob'
>>>D[’job'J = 'dev'
>>> D [' age' ] = 40
>>> D
{’age’  :  40, ’job’: ’dev', ’name’: ’Bob’}
>>> print (D [ ’ name' ])
Bob
```

Тут ми фактично застосовуємо ключі словника як імена полів у записі, який представляє когось. В інших додатках словники можуть використовуватися також для заміни операцій пошуку - індексування словника по ключу найчастіше виявляється найшвидшим способом пошуку в Python.

Пізніше буде показано, що ми можемо створювати словники також шляхом передачі імені типу dict або аргументів з ключовими словами (спеціальний синтаксис ім'я = значення у викликах функцій), або результату зв'язування разом послідовностей ключів та значень, отриманих під час виконання (наприклад, файлів) . Обидва наступні фрагменти створюють той же словник, що і в попередньому прикладі, і його еквівалентну літеральну форму { }, але другий фрагмент пов'язаний з меншим обсягом введення:

```
>>> bob1 = dict (name= ’Bob’ , job=’dev’, age-40)
>>> bob1
{’age’: 40, ’name’: ’Bob’, ’job’: ’dev’}
>>> bob2 = dict (zip ([’name ’ , ’job’, 'age'], [’Bob', ’dev’, 40]))
>>> bob2
{’job’: ’dev', ’name’: 'Bob', 'age': 40}
```

Зверніть увагу на те, як переплутується порядок зліва направо для ключів словника. Відображення не є позиційно впорядкованими, тому (якщо вам не пощастить) вони повернуться в порядку, що відрізняється від того, в якому ви їх набирали. Точний порядок може змінюватись в залежності від версії Python.

У попередньому прикладі ми використовували словник із трьома ключами для опису гіпотетичної особи. Однак припустимо, що інформація виявляється складнішою. Нехай нам потрібно зберігати ім'я та прізвище та кілька назв посад. У результаті ми маємо ще одне застосування вкладення об'єктів Python у дії. У наведеному далі словнику, одночасно представленому у вигляді літералу, міститься більш структурована інформація:

```
>>> rec = {'name' : {'first' : ’Bob' , 'last' : 'Smith’},
’jobs': ['dev’, 'mgr’],
'age': 40.5}
```

Ми знову маємо словник з трьома ключами ('name', 'j obs' і 'age'), але значення стали складнішими: вкладений словник для імені, що підтримує кілька частин, і вкладений список для назв посад, що підтримує багато назв та майбутнє розширення . Ми можемо отримувати доступ до компонентів цієї структури майже так, як раніше до матриці, заснованої на списках, але тепер більшість індексів є ключами у словниках, а не зміщеннями у списках:

```
>>> rec[’name’]
{'last': 'Smith', 'first': 'Bob'}
>>> rec['name’]['last’]
'Smith’
>>> rec [ ’ jobs' ]
```

```
['dev', 'mgr']
>>> rec ['jobs'][-1]
'mgr '
>>> rec [ ' jobs ' ] . append(' janitor ')
>>> rec
{'age': 40.5, 'jobs': ['dev', 'mgr', 'janitor'], 'name': {'last': 'Smith', 'first ' : 'Bob'}}
```

Зверніть увагу на те, як остання операція розширює вкладений список назв посад — оскільки список знаходиться в області пам'яті, окремої від словника, який містить його, він може вільно збільшуватися і зменшуватися.

Справжня причина, з якої показаний цей приклад, пов'язана з демонстрацією гнучкості основних типів даних Python. Ви можете помітити, що вкладення дозволяє будувати складні інформаційні структури безпосередньо та легко. Побудова схожої структури низькорівневою мовою на кшталт C була б стомлюючою і вимагала набагато більшого обсягу коду: довелося б планувати і оголошувати структури і масиви, заповнювати їх значеннями, пов'язувати всі разом і т.д. У Python все відбувається автоматично - виконання виразу створює цілу структуру вкладених об'єктів. Насправді це одна з головних переваг мов написання сценаріїв, подібних до Python.

У деяких загальних програмах при написанні коду ми завжди можемо знати, які ключі будуть у словниках під час виконання. Як обробляти такі сценарії, щоб уникнути помилок? Одне із рішень передбачає завчасну перевірку. Вираз перевірки членства у словнику, in, дозволяє вимагати існування ключа та організувати розгалуження залежно від результату за допомогою Python-оператора if.

```
>>> 'f ' in rec
False
>>> if not 'f ' in rec:
        print('missing')
missing
```

Крім перевірки за допомогою in існує ще багато інших способів уникнути звернення до неіснуючих ключів у словниках, що створюються:

метод get (умовний індекс зі стандартним варіантом); метод has_key з Python 2.X (аналог in, недоступний у Python 3.X); оператор try; і тернарний (що складається із трьох частин) вираз if/else, який по суті є оператором if, стислим в один рядок. Ось кілька прикладів:

```
>>> value = D,get('x' , 0)
>>> value
0
>>> value = D[’x’] if ’x’ in D else 0
>>> value
0
```

2.5. Кортежі

Об'єкт кортежу приблизно схожий на список, який не можна змінювати — кортежі є послідовностями подібно до списків, але вони незмінні подібно до рядків. Функціонально вони використовуються для представлення фіксованих колекцій елементів: наприклад компонентів специфічної дати в календарі. Синтаксично вони записуються в круглих, а не квадратних дужках і підтримують довільні типи, довільне вкладення та звичайні операції над послідовностями:

```
>>> T = (1, 2, 3, 4)
>>> len(T)
4
>>> T + (5, б)
(1, 2, 3, 4, 5, 6)
»> T[0]
1
```

Головна відмінність кортежів у тому, що з створення їх не можна змінювати, тобто. вони є незмінними послідовностями (одноелементні кортежі на кшталт наведеного нижче вимагають хвостової коми):

```
>>> T[0] = 2
```

TypeError: ’tuple’ object does not support item assignment

Подібно до списків і словників кортежі підтримують змішані типи та вкладення, але не збільшуються і не зменшуються, оскільки вони незмінні (круглі дужки, навколишні елементи кортежу, часто можна опускати, як зроблено тут; коми — це те, що фактично будує кортеж):

```
>>> T = 'spam', 3.0, [11, 22, 33]
>>> T[1]
3.0
>>> T[2][1]
22
>>> T.append(4)
AttributeError: 'tuple' object has no attribute 'append'
```

Отже, навіщо нам тип, що схожий на список, але підтримує менше операцій? Відверто кажучи, практично кортежі застосовуються загалом негаразд часто, як списки, але весь сенс у тому незмінності. Якщо ви передаєте колекцію об'єктів усередині своєї програми у вигляді списку, тоді він може бути модифікований будь-де; якщо ви використовуєте кортеж, змінити його не вдасться. Тобто кортежі забезпечують свого роду обмеження цілісності, що зручно в програмах, більші за ті, які ми писатимемо тут.

Завдання для виконання

1. В інтерактивному режимі обчислити значення числового виразу:
   1. $3 + \frac{1}{2} - 0,75$;
   2. $3 + \frac{1}{2}(0,75 + 3\frac{4}{5})$;
   3. $3 + \frac{1}{2} - \frac{0,75+3\frac{4}{5}}{1,15-4\frac{5}{6}}$;
   4. $\frac{1}{1+\sqrt{2}} + \frac{1}{\sqrt{2}+\sqrt{3}} + \frac{1}{\sqrt{3}+\sqrt{4}}$.
2. Скласти програму для обчислення значення виразу зі змінними*:
   1. $a + b - 3c$;
   2. $\frac{1}{2}a + b - 3c$;
   3. $a + \frac{b}{3c}$;
   4. $\frac{a+b}{3c}$;

5 $\frac{a+b}{3+c}$;

6 $\frac{a+b}{2(3+c)}$;

7 $a + b^2 - 3c$;

8 $\sqrt[5]{a} + b - 3c$;

9 $\sin a + \cos 3b - |7c|$;

10 $\mathrm{tg}a + \mathrm{ctg}3b$;

11 $e^a + \ln 3b + \pi$; 14) $e^a + \log_2 3b$.

*Примітка. У даній задачі (та в кожній наступній) значення змінних величин (вхідні дані) вводяться користувачем з клавіатури.

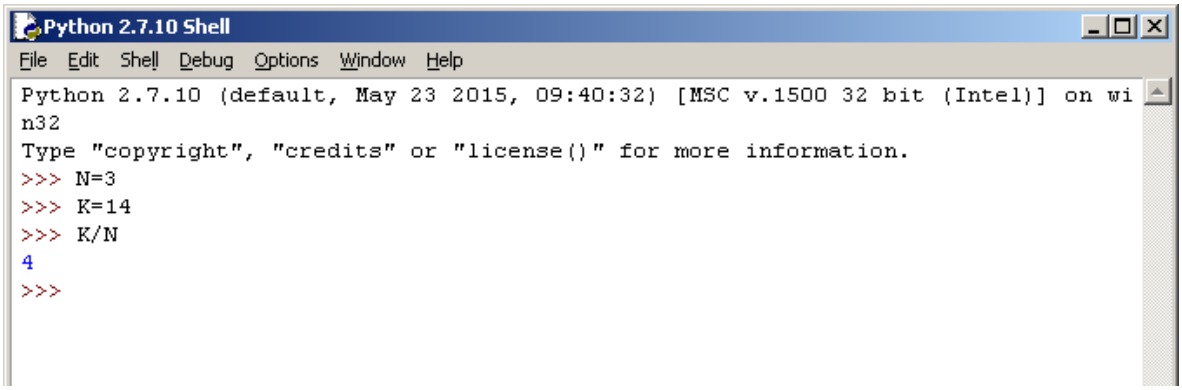
3. Скласти програму для обчислення середнього арифметичного та середнього геометричного двох натуральних чисел.
4. Скласти програму для обчислення суми, різниці, добутку та частки цифр двозначного числа. (Примітка. При обчисленні різниці та добутку розглянути всі можливі варіанти)
5. Скласти програму для обчислення об’єму та площі поверхні куба зі стороною a.
6. Скласти програму для обчислення рівнодійної сил, що діють на тіло масою m, якщо тіло рухається з прискоренням a.
7. Скласти програму для обчислення площі прямокутного трикутника, якщо відомі: а) катети; б) одиніз катетів та гіпотенуза.
8. Скласти програму для обчислення шляху, швидкості та прискорення руху в момент часу t, якщо рівняння руху визначається за формулою $S(t) = 3t^3 - 4t^2 + 7$.
9. Скласти програму для обчислення радіуса кола, вписаного у квадрат, та радіуса кола, описаногонавколо квадрату, якщо сторона квадрату a.
10. Скласти програму для обчислення загального опору трьох резисторів, що з’єднані: а) послідовно; б) паралельно.
11. *Скласти програму для обчислення значення виразу a^{15} (значення a вводиться з клавіатури), якщо допустимою є лише операція множення.

12. *Скласти програму для визначення кількості рулонів шпалер, що необхідно використати для оклеювання певної кімнати. (Примітка. Площею вікон та дверей знехтувати)
13. *Дано: натуральне число *n*. Визначити та вивести найменше парне число, що більше за *n*.
14. *Скласти програму для визначення кількості діб, що знадобиться для подолання відстані в *S* кілометрів, якщо швидкість руху становить *v* км/добу. (Примітки. Результат розв'язання задачі (кількість днів) має бути цілим числом, тоді як швидкість руху може набувати дійсних значень)
15. *Скласти програму для визначення чи є запис введеного натурального чотиризначного числа симетричним (напр., 3443 - симетричний, 3445 - несиметричний). Якщо число симетричне, надрукувати (вивести) 1, в інших випадках - довільне ціле число.
16. *Скласти програму для визначення найбільшого з двох цілих чисел (кожне з введених чисел належить проміжку від 1 до 1000).

Приклади для розв'язування завдання

1. N школярів ділять K яблук порівну, залишок що не ділиться залишається в корзинці. Скільки яблук дістанеться кожному школяру?

   Розв'язок



2. Дано натуральне число. Вивести його останню цифру

```
>>> N=179
>>> N%10
9
>>>
```

3. Дано тризначне число. Знайдіть суму його цифр.

```
>>> N=179
>>> a=N%10
>>> a
9
>>> N=N/10
>>> N
17
>>> b=N%10
>>> b
7
>>> c=N/10
>>> c
1
>>> a+b+c
17
```

Розділ 3. Оператор вибору варіанта дії на основі результатів перевірки

Формулюючи спрощено, оператор if мови Python вибирає дії для виконання. Разом зі своїм еквівалентом у вигляді виразу він є основним інструментом вибору Python і представляє великий обсяг логіки, якою володіє програма Python. До того ж це наш перший складовий оператор. Подібно до всіх складових операторів Python оператор if здатний містити інші оператори, включаючи додаткові if. Насправді Python дозволяє компонувати оператори у програмі послідовно (так що вони виконуються один за одним) і з довільною глибиною вкладення (щоб вони виконувались лише у разі задоволення певних умов, таких як ті, що застосовуються при виборі та в циклі).

Оператор if мови Python є типовим оператором if з більшості процедурних мов. Він приймає форму перевірки if, за якою слідує одна або більша кількість перевірок elif (“else if”) та фінальний необов'язковий блок else. З кожною перевіркою та частиною else пов'язаний блок вкладених операторів, розташований з відступом щодо рядка заголовка. Коли оператор if запускається, Python виконує блок коду, асоційований з першою перевіркою, яка дає справжній результат, або блок else, якщо усі перевірки мали помилкові результати.

```
if перевірка1:
     оператор1
elif перевірка2:
     оператор2
else:
     оператор3
```

Для демонстрації давайте подивимося на кілька прикладів оператора і f у роботі. Усі його частини необов'язкові крім початкової перевірки if та асоційованих з нею операторів. Таким чином, у найпростішому випадку інші частини опускаються:

```
>>> if 1:
```

```
        print(’true’)
true
```

Згадайте, що 1 – це булевське значення “істина” (як ви побачите пізніше, його еквівалентом є слово True), тому перевірка в операторі завжди проходить. Для обробки помилкового результату знадобиться додати частину else:

```
>>> if not 1:
          print(’true’)
 ...else:
          print('false’)
false
```

А ось приклад складнішого оператора if, що містить усі необов'язкові частини:

```
>>> x = ’killer rabbit’
>>> if x == ' roger ’ :
          print("shave and a haircut")
 ...elif x = ’bugs’ :
          print("what’s up doc?")
 ...else:
          print('Run away! Run away ?’)
Run away! Run away!
```

Наведений багаторядковий оператор тягнеться від рядка if до блоку, вкладеного всередину частини else. Коли він запускається, Python виконує оператори, вкладені всередину першої перевірки, яка дає справжній результат, або частина else, якщо всі перевірки показали помилкові результати (як у цьому прикладі). На практиці частини elifnelse можуть бути опущені, а в кожній частині допускається вказувати більше одного вкладеного оператора. Зверніть увагу, що слова if, elif та else зв'язуються один з одним за фактом вирівнювання по вертикалі з однаковими відступами.

Якщо ви використовували мови на кшталт C або Pascal, тоді вам цікаво буде дізнатися, що в Python відсутній оператор на кшталт switch або case, який вибирав би дію на основі значення змінної. Натомість множинне розгалуження зазвичай записується як серії перевірок if/elif, як у попередньому прикладі, і

іноді реалізується шляхом індексування словників чи пошуку у списках. Оскільки словники та списки можна динамічно створювати на стадії виконання, часом вони виявляються гнучкішими, ніж логіка if, жорстко закодована у сценарії:

```
>>> choice = 'ham'
>>> print({ 'spam' : 1.25,
     'ham': 1.99,
     'eggs': 0.99,
     'bacon': 1.10}[choice])
1.99
```

Хоча усвідомлення цього спочатку може вимагати деякого часу, показаний словник реалізує множинне розгалуження - індексація за ключом choice забезпечує перехід до одного з набору значень багато в чому подібно до оператора switch в C. Майже еквівалентний, але більш багатослівний оператор if мови Python міг би виглядати так:

```
>>>   if choice == 'spam' :  # Еквівалентний оператор if
          print(1.25)
. . . elif choice == 'ham' :
          print(1.99)
. . . elif choice == 'eggs' :
          print(0.99)
... elif choice == 'bacon':
          print(1.10)
. . . else:
          print('Bad choice')
1.99
```

У наведеному вище операторі if зверніть увагу на конструкцію else, яка призначена для обробки стандартного випадку, коли збіги ключа відсутні. Як пояснювалося в розділі 8, стандартні значення словників можна записувати за допомогою виразів in, викликів методу get або перехоплення винятків за допомогою оператора try, представленого в попередньому розділі. Ті самі методики можна застосовувати тут для кодування стандартної дії при множинному розгалуженні, заснованому на словнику. Як огляд у контексті

цього сценарію використання далі показано, як схема get працює зі стандартними значеннями:

```
>>> branch = { 'spam' : 1.25,
               'ham': 1.99,
               'eggs': 0.99}
>>> print (branch. get (’ spam' , ' Bad choice ’))
1.25
>>> print(branch.get(’bacon’, ’ Bad choice'))
Bad choice
```

Того ж результату можна досягти із застосуванням перевірки членства in в операторі if:

```
>>> choice = ’bacon’
>>>   if choice in branch:
          print(branch[choice])
. . . else:
          print(’Bad choice’)
Bad choice
```

Нарешті, оператор try пропонує узагальнений спосіб підтримки стандартних значень шляхом перехоплення та обробки винятків, які виникли б у протилежному випадку

```
>>>   try:
          print(branch[choice])
. . . except KeyError:
          print(’Bad choice’)
Bad choice
```

Розглянемо даний оператор докладніше на прикладі

```
>>> 100 / 0
Traceback (most recent call last):
  File "", line 1, in
    100 / 0
ZeroDivisionError: division by zero
```

Розберемо це повідомлення докладніше: інтерпретатор нам повідомляє про те, що він упіймав виняток і надрукував інформацію (Traceback (most recent call last)).

Далі - ім'я файлу (File ""). Ім'я порожнє, тому що ми в інтерактивному режимі, рядок у файлі (line 1);

Вираз, у якому сталася помилка (100/0).

Назва виключення (ZeroDivisionError) та короткий опис виключення (division by zero).

Зрозуміло, можливі й інші винятки:

```
>>> 2 + '1'
Traceback (most recent call last):
  File "", line 1, in
    2 + '1'
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> int('qwerty')
Traceback (most recent call last):
  File "", line 1, in
    int('qwerty')
ValueError: invalid literal for int() with base 10: 'qwerty'
```

У цих двох прикладах генеруються винятки TypeError та ValueError відповідно. Підказки дають нам повну інформацію про те, де породжений виняток, і з чим він пов'язаний.

Розглянемо ієрархію вбудованих у python винятків, хоча іноді вам можуть зустрітися й інші, оскільки програмісти можуть створювати власні винятки.

BaseException - базовий виняток, від якого беруть початок всі інші.

SystemExit – виняток, що породжується функцією sys.exit при виході з програми.

KeyboardInterrupt - породжується при перериванні програми користувачем (зазвичай поєднанням клавіш Ctrl+C).

GeneratorExit - породжується під час виклику методу close об'єкта generator.

Exception - а тут вже закінчуються повністю системні винятки (які краще не чіпати) і починаються прості, з якими можна працювати.

StopIteration - породжується вбудованою функцією next, якщо в ітератор більше немає елементів.

ArithmeticError - арифметична помилка.

FloatingPointError - породжується при невдалому виконанні операції з плаваючою комою. Насправді зустрічається нечасто.

OverflowError - виникає, коли результат арифметичної операції занадто великий для представлення. Не з'являється при звичайній роботі з цілими числами (оскільки python підтримує довгі числа), але може виникати в деяких інших випадках.

ZeroDivisionError - поділ на нуль.

AssertionError - вираз у функції assert хибний.

AttributeError - об'єкт не має атрибуту (значення або методу).

BufferError – операція, пов'язана з буфером, не може бути виконана.

EOFError – функція натрапила на кінець файлу і не змогла прочитати те, що хотіла.

ImportError - не вдалося імпортувати модуль або його атрибут.

LookupError - некоректний індекс чи ключ.

IndexError - індекс не входить до діапазону елементів.

KeyError - неіснуючий ключ (у словнику, множині або іншому об'єкті).

MemoryError - недостатньо пам'яті.

NameError - не знайдено змінної з такою назвою.

UnboundLocalError - зроблено посилання на локальну змінну до функцій, але змінна не визначена раніше.

OSError – помилка, пов'язана з системою.

BlockingIOError

ChildProcessError - невдача під час операції з дочірнім процесом.

ConnectionError - базовий клас для винятків, пов'язаних із підключеннями.

BrokenPipeError

ConnectionAbortedError

ConnectionRefusedError

ConnectionResetError

FileExistsError - спроба створення файлу чи директорії, яка вже існує.

FileNotFoundError – файл або директорія не існує.

InterruptedError - системний виклик перерваний вхідним сигналом.

IsADirectoryError - очікувався файл, але це директорія.

NotADirectoryError - очікувалася директорія, але це файл.

PermissionError - не вистачає прав доступу.

ProcessLookupError - цього процесу не існує.

TimeoutError - закінчився час очікування.

ReferenceError - спроба доступу до атрибуту зі слабким посиланням.

RuntimeError - виникає, коли виняток не підпадає під жодну з інших категорій.

NotImplementedError - виникає, коли абстрактні методи класу вимагають перевизначення в дочірніх класах.

SyntaxError - синтаксична помилка.

IndentationError - неправильні відступи.

TabError - змішування у відступах табуляції та пробілів.

SystemError - внутрішня помилка.

TypeError – операція застосована до об'єкта невідповідного типу.

ValueError - функція набуває аргументу правильного типу, але некоректного значення.

UnicodeError - помилка, пов'язана з кодуванням/розкодуванням unicode у рядках.

UnicodeEncodeError - виняток, пов'язаний з кодуванням unicode.

UnicodeDecodeError - виняток, пов'язаний з декодуванням unicode.

UnicodeTranslateError - виняток, пов'язаний з перекладом unicode.

Warning – попередження.

Тепер, коли ми підійшли до великих операторів на кшталт if, у цьому розділі пропонується огляд та розвиток ідей синтаксису. В цілому Python має простий синтаксис на основі операторів. Однак є кілька характеристик, про які слід знати:

• Оператори виконуються один за одним, якщо не вказано інше. Python зазвичай виконує оператори з файлу або вкладеного блоку в порядку з першого до останнього як послідовність, але оператори, подібні до if (а також цикли та винятки), змушують інтерпретатор здійснювати переходи в коді. Через те, що шлях Python через програму називається потоком управління, оператори на зразок if, які впливають на нього, часто називають операторами потоку управління.

• Кордони блоків та операторів визначаються автоматично. Як ви вже бачили, у Python відсутні будь-які дужки або обмежувачі “початку/кінця” у блоках коду; для групування операторів у вкладений блок використовуються їхні відступи щодо заголовка. Подібним чином оператори Python, як правило, не завершуються точкою з комою; натомість кінець рядка зазвичай означає закінчення оператора, записаного у цьому рядку. Як особливий випадок за допомогою спеціального синтаксису оператори можуть займати кілька рядків і розміщуватися в одному рядку.

• Складові оператори = заголовок + : + оператори з відступом. Усі складові оператори Python — із вкладеними операторами — випливають з одного шаблону: рядок заголовка, що завершується двокрапкою, за яким знаходиться один або більше операторів, зазвичай записаних з відступом щодо заголовка. Оператори з відступом називаються блоком (або іноді набором). В операторі if конструкції el if і else є частиною if, але також рядками заголовків з власними вкладеними блоками. Як особливий випадок блоки можуть перебувати в тому ж рядку, що і заголовок, якщо є неукладеним кодом.

• Порожні рядки, пробіли та коментарі зазвичай ігноруються. Порожні рядки є необов'язковими та ігноруються у файлах, але не в інтерактивній підказці, де вони завершують складові оператори. Прогалини всередині операторів та виразів майже завжди ігноруються (за винятком ситуації, коли вони знаходяться всередині рядкових літералів і коли застосовуються для відступу). Коментарі завжди ігноруються: вони починаються з символу # (не всередині рядкового літералу) і сягають кінця поточного рядка.

• Рядки документації ігноруються, але зберігаються та відображаються інструментами. Python підтримує додаткову форму коментарів, яку називають рядками документації, які на відміну від коментарів # запам'ятовуються під час виконання для інспектування. Рядки документації – це просто рядки, що відображаються у верхній частині файлів програм та ряду операторів. Python ігнорує їх вміст, але вони автоматично приєднуються до об'єктів під час виконання і можуть відображатися за допомогою інструментів документації, подібних до PyDoc.

Завдання для виконання

1. Дано число. Визначити, яке це число:
   1) додатне або від’ємне;
   2) додатне, від’ємне або нуль.
2. Дано число. Визначити, чи буде це число:

1) парним;

2) непарним;

3) кратним 3;

4) некратним 5;

5) парним і кратним 7;

6) непарним або некратним 7;

7) належати відрізку $[a;b]$ (передбачити введення значень a та b з клавіатури);

8) належати проміжку $(a;b) \cup [c;+\infty)$ (передбачити введення значень a, b та c з клавіатури);

9) задовольняти умову виду $|x| < a$ (використовуючи та не використовуючи математичні функції); 10) задовольняти умову виду $|x| > a$ (використовуючи та не використовуючи математичні функції).

3. Обчислити значення функції з урахуванням належності введеного значення аргументу області визначення функції:

1) $y = \quad x + 5$;

2) $y = x - 7$;

3) $y = \frac{1}{\sqrt{x+5}}$;

4) $y = \frac{1}{|x+5|}$;

5) $y = \frac{1}{x^7}$;

6) $y = \frac{1}{\sqrt{x+5}} + \frac{1}{x-7}$.

5. *Дано два числа. Вивести їх: а) у порядку зростання; б) у порядку спадання.

6. *Дано числа A, B та C. Визначити найбільше з них.

7. **Дано координати двох точок площини (напр. $(x_1;y_1)$, $(x_2;y_2)$). Визначити, чи лежать точки в одній координатній чверті. *Примітка. Всі координати точок мають бути відмінні від нуля.* Приклад

Вхідні дані	Відповідь

1 2 3 4	YES
1 2 -3 4	NO

8. **Дано три цілих числа *D* (день), *M* (місяць), *Y* (рік), що визначають існуючу дату року. Вивести дату, наступну за вказаною. *Примітка. Числа вводяться в окремих рядках.* Приклад

Вхідні дані	Відповідь
1 1 2015	2 1 2015
31 12 1998	1 1 1999

1. За номером місця у плацкартному вагоні визначити, яке це місце: верхнє або нижнє, в купе або бічне.
2. Чи можна з колоди, що має діаметр поперечного перерізу D, випиляти брус, що має квадратнийпереріз: а) площею *S*; б) периметром *P*?
3. Чи можна в квадратному залі площею S помістити круглу сцену радіусом *R* так, щоб від стіни до сцени був прохід не менше *K*?
4. Дано дійсні числа: *a*, *b*, *c*. Визначити, чи існує трикутник з такими довжинами сторін і, якщо існує, чи буде він прямокутним.

5. Є коробка зі сторонами: *A, B, C*. Визначити, чи пройде вона у двері з розмірами *M, K*.
6. Відомий РІК. Визначити, чи буде цей рік високосним, і до якого століття цей рік відноситься.
7. *Знайти дійсні корені квадратного рівняння. Коефіцієнти *a, b, c* вводяться з клавіатури. Якщо дійсних розв'язків немає, вивести повідомлення про це.
8. *Індекс маси тіла (ІМТ) — величина, що дозволяє оцінити відповідність маси тіла зросту людини.ІМТ обчислюється за формулою$I = \frac{m}{h^2}$, де *m* - маса тіла в кілограмах, *h* - зріст у метрах. Скласти програму для обчислення ІМТ та виведення повідомлення про відповідність маси тіла нормі згідно таблиці:

Індекс маси тіла	Відповідність між масою тіла та зростом людини
	Виражений дефіцит маси тіла
	Недостатня маса тіла
	Норма
	Надлишкова вага
	Ожиріння першого ступеня
	Ожиріння другого ступеня
	Ожиріння третього ступеня

9. *Користувач вводить число, що означає кількість років. Програма виводить це число й слово "рік",узгоджене з числом. Наприклад: "1 рік", "2 роки" "25 років".
10. *Відома грошова сума. Видати її купюрами, що є в наявності в 500, 200, 100, 50, 20 10, 5, 2, 1 грн.Кількість купюр має бути мінімальною.
11. **Поле шахової дошки визначається парою чисел (*a,b*), кожне з яких змінюється від 1 до 8, перше число задає номер стовпчика, друге – номер рядка. Задано дві комірки. Визначити, чи зможе шаховий король потрапити з першої комірки на іншу за один крок.

Вхідні дані

Дано 4 цілих числа від 1 до 8 кожне, перші дві задають початкову комірку, дві інші задають кінцеву комірку. Початкова та кінцева комірки не співпадають. Числа записані в окремих рядках.

Вихідні дані

Програма повинна вивести YES, якщо із початкової комірки ходом короля можна потрапити у другу, або NO в протилежному випадку.

Приклад

Вхідні дані	Відповідь
4 4 5 5	YES
1 3 1 5	NO

12. **В кожну крайню клітку квадратної дошки поставили по фішці. Чи могло виявитися, що виставлено рівно *k* фішок? (Наприклад, якщо дошка 2x2, то виставлено 4 фішки, а якщо 6x6 - то 20).

Вхідні дані

Вводиться одне натуральне число *k*.

Вихідні дані

Програма повинна вивести слово YES, якщо існує такий розмір дошки, на який буде виставлено рівно (не більше і не менше) *k* фішок, в іншому випадку - вивести слово NO. Приклад

Вхідні дані	Відповідь
20	YES
13	NO

1. Є два числа X та Y. Збільшити X на 1, якщо він менше Y.

```
>>> x=4
>>> y=7
>>> if x<y:
        x = x + 1

>>> x
5
```

2. Написати програму, яка за введеною температурою повітря видає текст “Хороша погода”, якщо температура більше 10, й “Погана погода” в протилежному випадку

```
>>> t=input('Введіть температуру в градусах: ')
Введіть температуру в градусах: 4
>>> if t<10:
        s='Погана погода'
else:
        s='Хороша погода'


>>> print s
Погана погода
```

3. Написати програму введення оцінки P та виводу тексту “Відмінно!”, якщо P=5, “Добре!”, якщо P=4, та “Ледащо!!!”, якщо P<4

```
>>> P=input('Ваша оцінка: ')
Ваша оцінка: 3
>>> if P==5:
        s='Молодець!'
elif P==4:
        s='Добре!'
else:
        s='Ледащо!!!'


>>> print s
Ледащо!!!
```

Розділ 4 Циклічні конструкції

У цьому розділі ми розглянемо дві головні циклічні конструкції мови — операторів, які повторюють дію знову і знову. Перший з них, оператор while, схиляє спосіб написання універсальних циклів. Другий, оператор for, призначений для проходу елементами в послідовності або в іншому об'єкті, що ітерується, і виконання блоку коду для кожного елемента.

По ходу справи ми також досліджуємо менш помітні оператори, які застосовуються всередині циклів, такі як break і continue, а також розкриємо ряд вбудованих функцій, які часто використовуються з циклами, у тому числі range, zip і map.

Хоча аналізовані в розділі оператори while і for є головним синтаксисом, пропонованим для кодування повторюваних дій, у Python існують інші процеси і концепції організації циклів. У зв'язку з цим історія про ітерацію продовжується в наступному розділі, де ми будемо досліджувати пов'язані ідеї протоколу ітерації Python (застосовуваного циклом for) та спискових включень (близький родич циклу for) і навіть більш екзотичні інструменти ітерації, такі як генератори, filter та reduce. Але поки що давайте займемося простими речами.

2.1. While

Оператор while – найуніверсальніша конструкція для ітерацій у мові Python. Висловлюючись простими термінами, він багаторазово виконує блок операторів (зазвичай з відступом) до тих пір, поки перевірка в заголовку оцінюється як справжнє значення. Це називається "циклом", тому що керування продовжує повертатися до початку оператора, поки перевірка не дасть хибне значення. Коли результат перевірки стає хибним, керування переходить на оператор, наступний після блоку while. Сукупний ефект у тому, що тіло циклу виконується багаторазово, поки перевірка у заголовній частині

дає справжнє значення. Якщо перевірка оцінюється в хибне значення від початку, тоді тіло циклу будь-коли виконається і оператор while пропускається.

У своїй найскладнішій формі оператор while складається з рядка заголовка з виразом перевірки, тіла з однією або великою кількістю оператором з відступами та необов'язковою частиною else, яка виконується, якщо керування залишає цикл, а оператор break не зустрівся. Python продовжує оцінювати вираз перевірки в рядку заголовка і виконує оператори, вкладені в тіло циклу, доки перевірка не поверне хибне значення:

```
while перевірка:
     оператори
else:
     операторы
```

З метою ілюстрації давайте подивимося на кілька простих циклів, коли в дії. Перший, який складається з оператора print, вкладеного в цикл while, лише нескінченно виводить повідомлення. Згадайте, що True це спеціальна версія цілого числа 1 і завжди позначає булевське справжнє значення; оскільки перевірка завжди дає істину, Python продовжує виконання тіла до нескінченності або доти, доки ви не зупините його виконання. Поведінка такого виду зазвичай називається нескінченним циклом - по правді кажучи, він не вічний, але вам може знадобитися натиснути комбінацію клавіш <Ctrl+C>, щоб примусово закінчити його роботу:

```
>>> while True:
     print('Type Ctrl-C to stop me! ’)
```

У наступному прикладі проводиться відкидання першого символу рядка до тих пір, поки він не стане порожнім і тому помилковим. Цілком зазвичай перевіряти об'єкт безпосередньо замість більш багатослівного еквівалента (while x != ’ ’ :).

Зверніть увагу на ключовий аргумент end=' ', застосовуваний тут для розміщення всіх висновків у тому ж рядку з поділом їх пробілами.

У нижченаведеному коді проводиться підрахунок від значення a до значення b, не включаючи його. Пізніше ми побачимо більш легкий спосіб такого підрахунку за допомогою циклу для мови Python і вбудованої функції range:

```
>>> a=0; b=10
>>> while a < b:
... print(a, end=’ ’)
... a += 1
0 1 2 3 4 5 6 7 8 9
```

Тепер, коли було продемонстровано кілька циклів Python у дії, настав час поглянути на два простих оператори, які досягають своїх цілей, тільки коли вкладені всередині циклів — break і continue. Займаючись тут такими незвичайними операторами, ми також розглянемо конструкцію else циклу, тому що вона переплетена з break, і порожній оператор-заповнювач pass (який не прив'язаний до циклів, але відноситься до загальної категорії простих однослівних операторів). Вказані інструменти Python описані нижче:

- break – переходить за межі найближчого циклу, що укладає (після всього оператора циклу).
- continue – переходить на початок найближчого циклу, що укладає (на рядок заголовка циклу).
- pass – загалом нічого не робить: це порожній оператор-заповнювач.
- else – виконується тоді і лише тоді, коли відбувається нормальний вихід із циклу (тобто без виконання оператора break).

Оператор break викликає негайний вихід із циклу. Оскільки при його досягненні код, який знаходиться за ним у тілі циклу, не виконується, за рахунок увімкнення break іноді можна уникнути вкладення. Наприклад, нижче представлений простий інтерактивний цикл (варіант більшого прикладу, який наводився в розділі 10), де за допомогою input вводяться дані; коли користувач у відповідь на запит імені вводить слово stop, відбувається вихід із циклу:

```
>>> while True:
... name = input('Enter name:’)
```

```
... if name == ' stop ’ : break
... age = input (’ Enter age: ’)
... print(’Hello’, name, ’=>’, int(age) ** 2)
Enter name:bob
Enter age: 40
Hello bob => 1600
Enter name:sue
Enter age: 30
Hello sue => 900
Enter name:stop
```

Оператор continue викликає негайний перехід початку циклу. Іноді він також дозволяє уникнути вкладення операторів. У наведеному прикладі оператор continue використовується для пропуску виведення непарних чисел. Код виводить усі парні числа, які менші за 10 і більші або рівні 0. Згадайте, що 0 означає брехню, а % — операцію отримання залишку від поділу (модуля), тому даний цикл робить зворотний відлік до 0, пропускаючи числа, які не є множниками 2 - він виводить 8 6 4 2 0:

```
>>> x = 10
>>> while x:
... x = x-1
... if x % 2 != 0: continue
print(x, end=’ ')
```

У поєднанні з конструкцією else циклу оператор break часто дозволяє усунути потребу у прапорах стану пошуку, які використовуються іншими мовами. Скажімо, наступний фрагмент коду визначає, чи є позитивне ціле число у простим, за рахунок пошуку співмножників більше 1:

```
x = y // 2
while x > 1:
     if y % x == 0:
          print (y,  ’has  factor', x)
          break
     x -= 1
else:
     print (y,  'is prime’)
```

Замість установки прапора, призначеного для перевірки, чи потрібно виходити з циклу, в коді застосовується оператор break, коли знайдено змножувач. У результаті можна припустити, що конструкція else циклу буде виконуватися тільки якщо співмножники не знайдені; попадання на break означає, що число просте. Виконайте код крок за кроком, щоб подивитися, як він працює.

Конструкція else циклу також виконується, якщо тіло циклу ніколи не виконувалося, тому що в цьому випадку не виконується і оператор break; у циклі while подібне відбувається, коли перевірка в заголовку від початку дає хибне значення. Таким чином, у попередньому прикладі все одно буде виводитися повідомлення is prime, якщо x спочатку менше або дорівнює 1 (тобто коли дорівнює 2).

2.2. For

Цикл for є універсальним ітератором у Python: він може проходити елементами в будь-якій упорядкованій послідовності або в іншому об'єкті, що ітерується. Оператор for працює на рядках, списках, кортежах та інших вбудованих об'єктах, що ітеруються, а також на нових об'єктах, що визначаються користувачем, які ми пізніше навчимося створювати за допомогою класів.

Цикл для мови Python починається з рядка заголовка, де вказується мета (або цілі) присвоєння поряд з об'єктом, яким потрібно зробити прохід. Після заголовка знаходиться блок операторів (зазвичай із відступами), який необхідно повторювати:

```
for ціль in об’єкт:
     оператори
else:
     оператори
```

Коли Python запускає цикл for, він присвоює цілі елементи об'єкта, що ітерується, по черзі і виконує для кожного тіло циклу. Всередині тіла циклу

ціль присвоювання зазвичай використовується для посилання на поточний елемент у послідовності, як би мета була курсором, що проходить через послідовність.

Ім'я, яке застосовується як мета присвоювання в рядку заголовка for, зазвичай є (можливо, новою) змінною всередині області видимості, де знаходиться оператор for. Ім'я не відрізняється якоюсь унікальністю; його навіть можна змінювати всередині тіла циклу, але воно буде автоматично встановлюватися в наступний елемент послідовності, коли керування знову повернеться до початку циклу. Після циклу ця змінна, як правило, як і раніше посилається на останній відвіданий елемент, яким буде останній елемент у послідовності, якщо тільки не відбувся вихід із циклу за допомогою оператора break.

Оператор for також підтримує необов'язковий блок else, який працює точно як у циклі while - він виконується, якщо вихід із циклу здійснюється без допомоги оператора break (тобто коли були відвідані всі елементи послідовності). Представлені раніше оператори break та continue у циклі for також працюють аналогічно циклу while.

Як згадувалося раніше, цикл for може проходити об'єктом послідовності будь-якого виду. У нашому першому прикладі ми будемо надавати ім'я x кожен із трьох елементів по черзі, зліва направо, і для кожного з них виконуватиметься оператор print. Всередині оператора print (тіло циклу) ім'я x посилається на поточний елемент у списку:

```
>>> for x in ["spam”, "eggs”, "ham”]:
... print(x, end=’ ’)
spam eggs ham
```

У наступних двох прикладах обчислюється сума та добуток всіх елементів у списку:

```
>>> sum = 0
>>> for x in [1, 2, 3, 4] :
... sum = sum + x
>>> sum
```

```
10
>>> prod = 1
>>> for item in [1, 2, 3, 4] : prod * =  item
>>> prod
24
```

У циклі for працює будь-яка послідовність, так як він є універсальним інструментом. Скажімо, цикли for працюють на рядках та кортежах:

```
>>> S = ''lumberjack''
>>> T = ("and", ''I'm'', ''okay")
>>> for x in S: print (x, end=' ')
l u m b e r j a c k
>>> for x in T: print(x, end=' ')
and I'm okay
```

При проході по послідовності кортежів сама змінна циклу буде кортежем цілей. Це всього лише ще один випадок привласнення кортежів, що розпаковує. Згадайте, що цикл for привласнює цілі елементи в об'єкті послідовності і присвоювання працює однаково всюди:

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T:
... print(a, b)
1 2
3 4
5 6
```

Перший прохід циклу подібний до написання (a, b) = (1,2), другий прохід - (a, b) = (3,4) і т.д. Сукупним ефектом є автоматичне розпакування поточного кортежу кожної ітерації.

Така форма зазвичай застосовується у поєднанні із викликом вбудованої функції zip, яку ми побачимо пізніше на чолі при реалізації паралельних обходів. Вона також регулярно зустрічається разом з базами даних SQL у Python, коли таблиці результатів запиту повертаються у вигляді послідовностей, подібних до використаного тут списку — зовнішній список є таблицею бази даних, вкладені кортежі є рядками всередині таблиці, а привласнення кортежів витягує стовпці.

Кортежі в циклах for також стають корисними при ітерації відразу за ключами та значеннями у словниках із застосуванням методу items замість проходу в циклі за ключами та індексації для вилучення значень вручну:

```
>>> D = {'a' : 1, 'b' : 2, 'c' : 3}
>>> for key in D:
... print(key, '=>', D[key])
a => 1
c => 3
b => 2
>>> list(D.items ())
[('a', 1), ('c', 3), ('b', 2)]
>>> for (key, value) in D.items():
... print (key, '=>' , value)
a => 1
c => 3
b => 2
```

Але кортежі в заголовку циклу заощаджують додатковий крок при ітерації за послідовністю послідовностей. У циклі for можуть автоматично розпаковуватися також і вкладені структури даних:

```
>>> ((a, b) , c) = ((1, 2) , 3)
>>> a, b, c
(1, 2, 3)
>>> for ((a, b) , c) in [ ((1, 2) , 3) , ((4, 5) , 6) ] :
... print (a, b, c)
1 2 3
4 5 6
```

Тим не менш, навіть це не є особливим випадком — цикл for просто виконує на кожній ітерації різновид присвоювання, який робився перед ним. Таким способом може бути розпакована будь-яка структура з вкладеними послідовностями просто через настільки універсальну природу присвоєння послідовностей.

```
>>> for ((a, b) , c) in [ ([1, 2] , 3) , ['XY' , 6] ] :
... print(a, b, c)
1 2 3
```

```
X Y 6
```

Через те, що Python 3.X послідовність може бути присвоєна більш загальному набору імен з позначеним зірочкою ім'ям для збору безлічі елементів, той же синтаксис може застосовуватися для вилучення частин вкладених послідовностей в циклі for:

```
>>> a, * b,  c = (1, 2, 3, 4)
>>> a, b, c
(1, [2, 3], 4)
>>> for (a, * b,  c) in [ (1, 2, 3, 4) , (5, 6, 7, 8) ] :
... print (a, b, c)
1 [2, 3] 4
5 [6, 7] 8
```

Давайте тепер поглянемо на цикл for, який складніший за ті, що ми бачили досі. У наведеному нижче прикладі ілюструється вкладення операторів та конструкція else циклу for. Маючи список об'єктів (items) та список ключів (tests), код шукає кожен ключ у списку об'єктів та повідомляє про результат пошуку:

```
>>> items = ("aaa", 111, (4, 5), 2.01]
>>> tests = [(4, 5) , 3.14]
>>>
>>> for key in tests:
for item in items:
...  if item == key:
...       print (key, ’’was found”)
          break
else:
...  print(key, "not found!")
(4, 5) was found
3.14 not found!
```

Оскільки у вкладеному операторі if виконується break, коли збіг виявлено, у конструкції else циклу можна припустити, що й досягнуто, то пошук зазнав невдачі. Зверніть увагу на вкладення. Після запуску коду одночасно виконуються два цикли: зовнішній цикл переглядає список ключів,

а внутрішній список елементів для кожного ключа. Вкладення конструкції else циклу критично важливе; вона зміщена той самий рівень, як і рядок заголовка внутрішнього циклу for, тому асоціюється з внутрішнім циклом, але з оператором if чи зовнішнім циклом for.

Приклад є ілюстративним, але його код можна спростити, якщо використовувати операцію in для перевірки членства. Через те, що операція in неявно переглядає об'єкт у пошуку збігу (принаймні логічно), вона замінює внутрішній цикл:

```
for key in tests:
...  if key in items:
          print(key, "was found")
     else:
          print(key, "not found?")
(4, 5) was found
3.14 not found!
```

Загалом задля стислості та високої продуктивності розумно доручати Python виконання якомога більшого обсягу роботи (що ілюструється у наведеному вище рішенні).

Щойно досліджений цикл for відноситься до найбільшої категорії циклів з підрахунком. Він зазвичай простіше в написанні і часто виконується швидше, ніж while, тому є першим інструментом, до якого ви повинні звертатися щоразу, коли необхідно проходити через послідовність або інший об'єкт, що ітерується. Насправді, як правило, ви повинні чинити опір спокусі підраховувати будь-що в Python — його ітераційні засоби автоматизують більшу частину роботи, що виконується для проходу по колекціях у мовах нижчого рівня на кшталт C.

Проте існують ситуації, коли потрібно виконувати прохід спеціалізованими способами. Скажімо, якщо потрібно відвідати кожен другий чи кожен третій елемент у списку, або принагідно змінювати список? Як щодо обходу більше однієї послідовності паралельно у тому самому циклі for? Що якщо потрібна також індексація?

Ви завжди можете записувати такі специфічні ітерації за допомогою циклу while та ручної індексації, але Python пропонує набір вбудованих функцій, які дозволяють спеціалізувати ітерацію у циклі for.

• Вбудована функція range (доступна, починаючи з Python O.X) виробляє серію цілих чисел, що послідовно зростають, які можуть використовуватися як індекси в циклі for.

• Вбудована функція zip (доступна, починаючи з версії Python 2.0), повертає серію кортежів з паралельних елементів, які можуть застосовуватися для обходу безлічі послідовностей у циклі for.

• Вбудована функція enumerate (доступна, починаючи з версії Python 2.3) генерує значення та індекси елементів в об'єкті, що ітерується, так що вести рахунок вручну не доведеться.

Перша пов'язана з циклами функція, range, насправді є універсальним інструментом, який можна використовувати у різноманітних контекстах. Хоча функція range найчастіше буде застосовуватися для генерації індексів у циклі for, ви можете її використовувати будь-де, коли потрібна серія цілих чисел. Python 2.X функція range створює фізичний список; в Python З.X функція range є об'єктом, що ітерується, який генерує елементи за запитом, тому для відображення відразу всіх результатів виклик range знадобиться помістити всередину виклику list:

```
>>> list(range(5)), list(range(2, 5)), list(range (0, 10, 2))
([0, 1, 2, 3, 4] , [2, 3, 4], [0, 2, 4, 6, 8])
```

З одним аргументом range генерує список цілих чисел, починаючи з нуля та закінчуючи, але не включаючи значення аргументу. У разі передачі двох аргументів перший вважається нижнім кордоном. Необов'язковий третій аргумент може задавати крок", коли він зазначений, Python додає обраний крок до кожного наступного цілого числа в результаті (за замовчуванням крок дорівнює +1). За бажанням діапазони можуть бути непозитивними та незростаючими.

Результати роботи range виявляються найзручнішими всередині циклів for. Насамперед вони надають спосіб для повторення дії зазначену кількість разів. Наприклад, щоб вивести три рядки, за допомогою range генерується відповідна кількість цілих чисел:

```
>>> for i in range (3) :
... print(i 'Pythons')
0 Pythons
1 Pythons
2  Pythons
```

Зверніть увагу, що цикли for в Python 3.X автоматично отримують результати з range, тому виклик list тут застосовувати не потрібно (у Python 2.X ми отримуємо тимчасовий список, якщо замість цього не викликали xrange).

Методика, що розглядається в цьому розділі, розширює область дії циклу. Як ви бачили, вбудована функція range дозволяє обходити послідовності за допомогою циклу for у неповній манері. В тому ж дусі вбудована функція zip дає можливість застосовувати цикли for для перегляду безлічі послідовностей паралельно - без суміщення в часі, але протягом того самого циклу. У базовому вигляді функція zip приймає одну або більше послідовностей як аргументи і повертає серію кортежів, що об'єднують у пари паралельні елементи із зазначених послідовностей. Припустимо, що ми працюємо з двома списками (можливо, списком імен та списком адрес, що узгоджуються за позиціями):

```
>>> L1 = [1,2,3,4]
>>> L2 = [5,6,7,8]
>>> zip(Ll, L2)
<zip object at 0x026523C8>
>>> list(zip(Ll, L2))
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

Такий результат може бути корисним і в інших контекстах, але в поєднанні з циклом він підтримує паралельні ітерації:

```
>>> for (x, y) in zip(L1, L2) :
```

```
... print(x, y, ’ —’ , x+y)
1 5 — 6
2 6 — 8
3 7 — 10
4 8 — 12
```

Тут ми проходимо результатом виклику zip, тобто. за парами елементів, витягнутих із двох списків. Зверніть увагу, що цей цикл for знову застосовує представлену раніше форму привласнення кортежів для розпакування кожного кортежу в результаті zip. При першому проході виходить так, ніби ми виконували оператор присвоювання (x, y) = (1, 5).

Остання допоміжна функція, що розглядається, призначена для підтримки подвійного режиму використання. Раніше ми обговорювали застосування range для генерації зсувів елементів у рядку, а не самих елементів цих зсувів. Тим не менш, у ряді випадків нам необхідно те й інше: елемент для використання та усунення у міру просування. Традиційно таке завдання вирішувалося за допомогою простого циклу for, який також вів лічильник поточного усунення:

```
>>> S = ’ spam'
>>> offset = 0
>>> for item in S:
... print(item, ’appears at offset', offset)
... offset += 1
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

Прийом працює, але в недавніх випусках Python 2.X і 3.X (починаючи з версії Python 2.3) подібну роботу робить нова вбудована функція на ім'я enumerate - вона забезпечує цикли лічильником "безкоштовно", не змушуючи приносити в жертву простоту автоматичної ітерації:

```
>>> S = 'spam'
>>> for (offset, item) in enumerate(S) :
... print(item, 'appears at offset', offset)
```

```
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

Функція enumerate повертає генераторний об'єкт. Такий об'єкт має метод, який викликається вбудованою функцією next, який на кожному проході циклу повертає кортеж {індекс, значення}. Цикл for проходить по цих кортежах автоматично, що дозволяє розпаковувати їх значення за допомогою надання кортежів майже так, як ми робили для zip.

Завдання для виконання

1. Вивести а) перші 20; б) перші *N* натуральних чисел.
2. Вивести на екран таблицю множення
3. Вивести на екран шкалу термометра від −40 до +40 градусів з кроком 10 градусів у вигляді:
   −40
   −30
   ...
   +30
   +40
4. Вивести таблицю значень функції $y = \cos(2x + 5)$, якщо значення аргументу належать проміжку [−100;100] і змінюються з кроком 5,5.
5. Розрахувати суму *N* введених чисел. Число *N* вводиться з клавіатури.
6. Розрахувати кількість парних чисел із *N* введених.
7. Обчислити значення факторіалу числа *N*. Число *N* вводиться з клавіатури.
8. Розрахувати добуток всіх додатних із *N* введених чисел.
9. Розрахувати середньотижневу температуру, якщо користувач вводить з клавіатури середньодобовутемпературу за кожен день.
10. Програміст Петрик П’яточкін страждає на безсоння. Щоб швидше заснути, він рахує слоненят від 1 до 10. Одного разу він зрозумів, що процес

рахування слонів можна автоматизувати, якщо написати спеціальну програму на Python. Допоможіть йому в цьому. Результат роботи програми може мати такий вигляд:

1

2

...

10

11. Після успішного тестування бета-версії програми для рахування слоненят Elephant Counter (див. попередню задачу) програміст Петрик П’яточкін отримав безліч звернень від користувачів, що страждають на більш важкі форми безсоння, із проханням збільшити кількість підтримуваних слоненят із десяти до 50, 100 та 1000. Щоб задовольнити всі можливі вподобання користувачів, було прийнято рішення дозволити користувачам самостійно вводити кількість слоненят для підрахунку. Напишіть вдосконалену версію Elephant Counter v.2.0, яка б відповідала цій вимозі.
12. Для збільшення кількості потенційних користувачів успішний стартапер Петрик П’яточкін (див. попередню задачу) вирішив розділити проект Elephant Counter на дві гілки: Starter Edition і Premium Deluxe Edition. На прохання користувачів, що разом із безсонням страждають на відсутність уяви (як і автор цих одноманітних задач), у версії Premium Deluxe Edition мали з’явитися реалістичні візуальні спецефекти. Залучений до проекту дизайнер створив ескіз, наведений нижче. Напишіть програму Elephant Counter v.3.0 Premium Deluxe Edition.

    Примітка. Для виведення зображення використовуйте символи “/”, “–”, “\”, “_” й т. п. Зверніть увагу, що для виведення символу “\” (backslash, зворотний слеш) слід писати “\\”, оскільки backslash використовується для позначення службових символів.

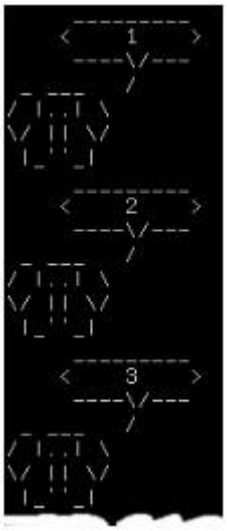


Рис. 1.

13. *Після приголомшливого успіху Elephant Counter v.3.0 Premium Deluxe Edition (див. попередню задачу) виявилося, що користувачі базової версії Starter Edition, які не побажали витрачати кошти на оновлення до Premium Deluxe Edition, незадоволені інтерфейсом програми, оскільки вона виводить лише набір чисел, що, на відміну від Premium Deluxe Edition, не дає змогу ідентифікувати істот, що підлягають підрахунку. Зокрема, Elephant Counter Starter Edition мало чим відрізняється від свого лютого конкурента Sheep Counter, що пропонує рахувати вівці перед сном. Топ-менеджер проекту Петрик П’яточкін запропонував створити версію Elephant Counter v.3.0 Starter Edition із таким інтерфейсом: 1 слоненя
2 слоненяти
3 слоненяти
4 слоненяти
5 слоненят
...
Напишіть цю версію програми. Врахуйте, що слово “слоненя” має виводитися в правильному відмінку.
14. *До контакт-центру проекту Elephant Counter v.3.0 Premium Deluxe Edition (див. попередню задачу) почали надходити звернення від стурбованих користувачів, які звинувачували розробників у зловживаннях і недостатньо

наочному підрахунку слоненят. Обурені користувачі вимагали повного прозорого перерахунку слонів. На вимогу голови ради директорів Петрика П’яточкіна відділ дизайну створив ескіз нової концепції, що наведений нижче. Напишіть програму Elephant Counter v.4.0 Premium Deluxe Edition, результат роботи якої подібний до наведеного ескізу.

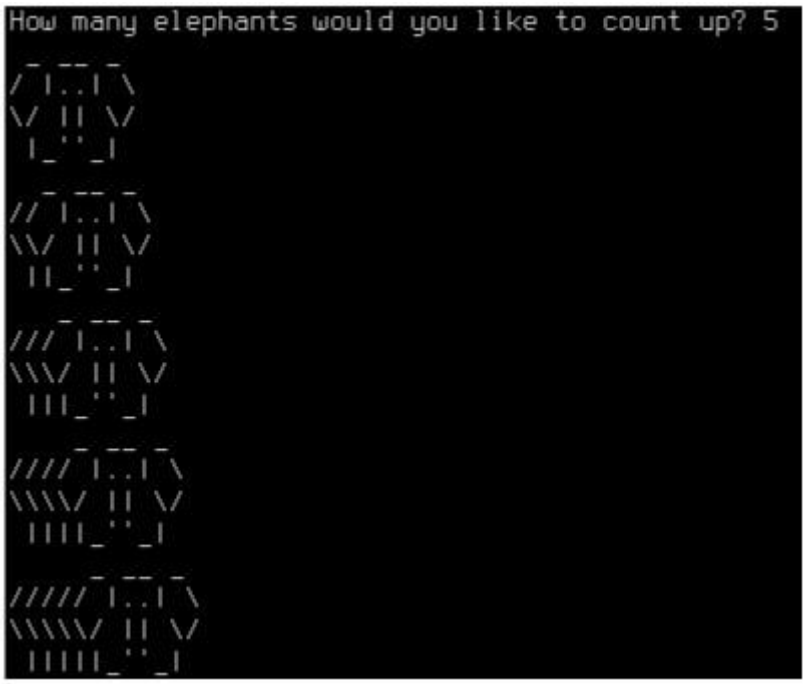


Рис. 2.

15. *Після виходу проекту Elephant Counter (див. попередню задачу) на міжнародний ринок з’ясувалося,що в Сполучених Штатах програма користується попитом лише серед прихильників республіканської партії, тоді як демократи жадають рахувати віслюків. В Індії проект знаходився на межі повного провалу, позаяк рахування слонів сприймалося як зневага до бога Індри та погрожувало посухами й іншими катаклізмами. Через те рада директорів прийняла рішення створити локалізовані версії програми з різними видами тварин. Напишіть програму Elephant Counter v.5.0 Ultimate International Premium Deluxe Edition, що дозволяє користувачеві обирати принаймні з двох тварин (на ваш вибір).
16. *З нагоди виходу ювілейної золотої збірки всіх версій програми Elephant Counter (із власноручнимавтографом розробника) створіть меню, що дозволить запустити потрібну версію за вибором користувача. Користувач

робить вибір шляхом вводу з клавіатури номера пункту меню. Після закінчення роботи обраної версії Elephant Counter меню виводиться повторно й користувачеві знову пропонується зробити вибір. Так відбувається доти, доки користувач не введе 0 (нуль), що означає вихід з програми. Зовнішній вигляд меню придумайте самостійно на власний розсуд. Наприклад, воно може виглядати так:

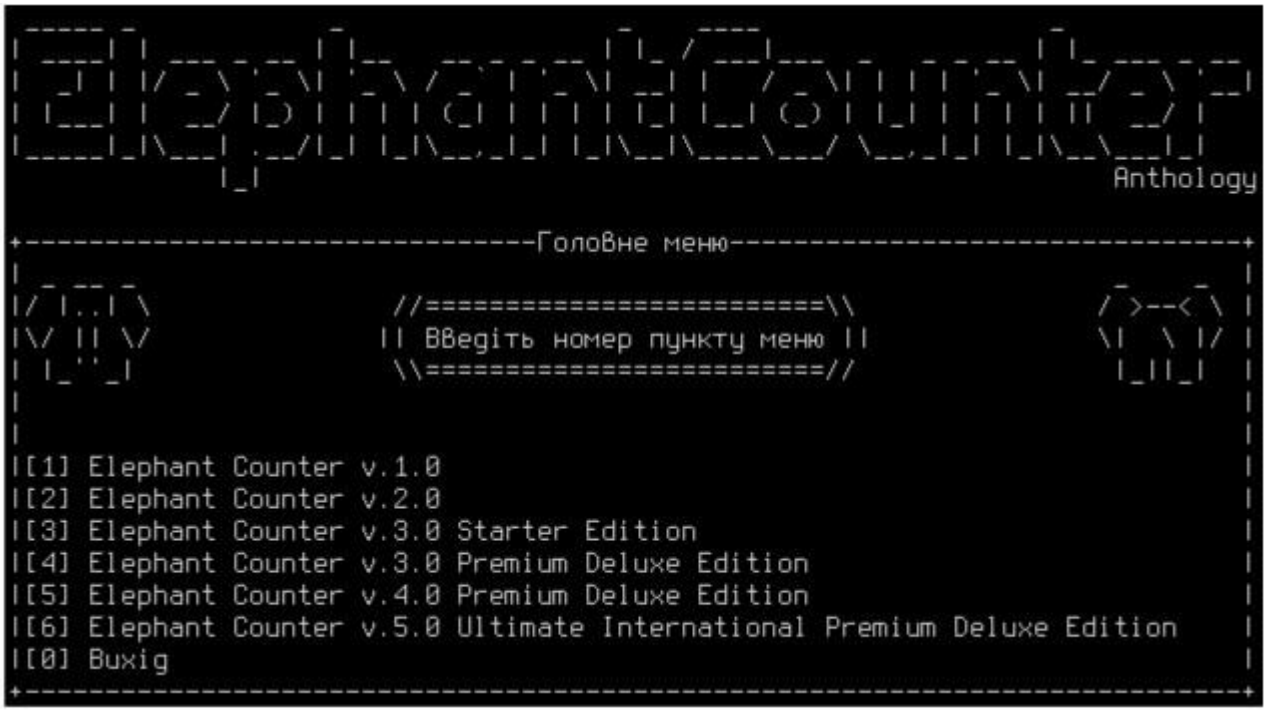


Рис. 3.

17. **Натуральне *n*-значне число є числом Армстронга, якщо сума його цифр, піднесених до *n*–го ступеня, дорівнює самому числу ($153 = 1^3 + 5^3 + 3^3$). Одержати всі числа Армстронга для $n = 2,3,4$.
18. **Шеф-кухар вирішив влаштувати у ліцеї день поваги до кухарів. Для цього він приготував ліцеїстам *n* надзвичайно смачних котлет і постановив, що перший ліцеїст, який прийшов скуштувати кухарську страву повинен отримати найбільшу кількість смачних котлет, а кожен наступний – строго менше, ніж попередній (кухарю дуже не подобається, коли до обіду, що він приготував спізнюються і той буде остигати). Звичайно, введення такого правила істотне свавілля в числі котлет, що будуть отримані черговим ліцеїстом. Наприклад, 6 котлет в результаті можуть бути розподілені по одній з наступних чотирьох схем:

1 3 + 2 + 1 (три котлети першому ліцеїсту, дві – другому і одну – третьому);

2 4 + 2;

3 5 + 1;

4 6 (всі котлети з'їдає щасливчик, який прийшов перший).

Напишіть програму, що визначає, якою кількістю різних способів кухар може розподілити котлети серед ліцеїстів.

Вхідні дані

Вводиться одне ціле число n – кількість підготовлених кухарем котлет ($0 < n < 200$).

Вихідні дані

На екран виводиться одне ціле число, рівне кількості можливих розподілів котлет.

20. **Задано натуральне число N. Знайти найменше та найбільше число, яке складається з тих самих цифр та у такій самій кількості, що і N.

Вхідні дані

З клавіатури вводиться число N ($1 < N < 2000000000$).

Вихідні дані

Виводиться два числа в одному рядку – найменше число, а через пропуск – найбільше число.

Приклад

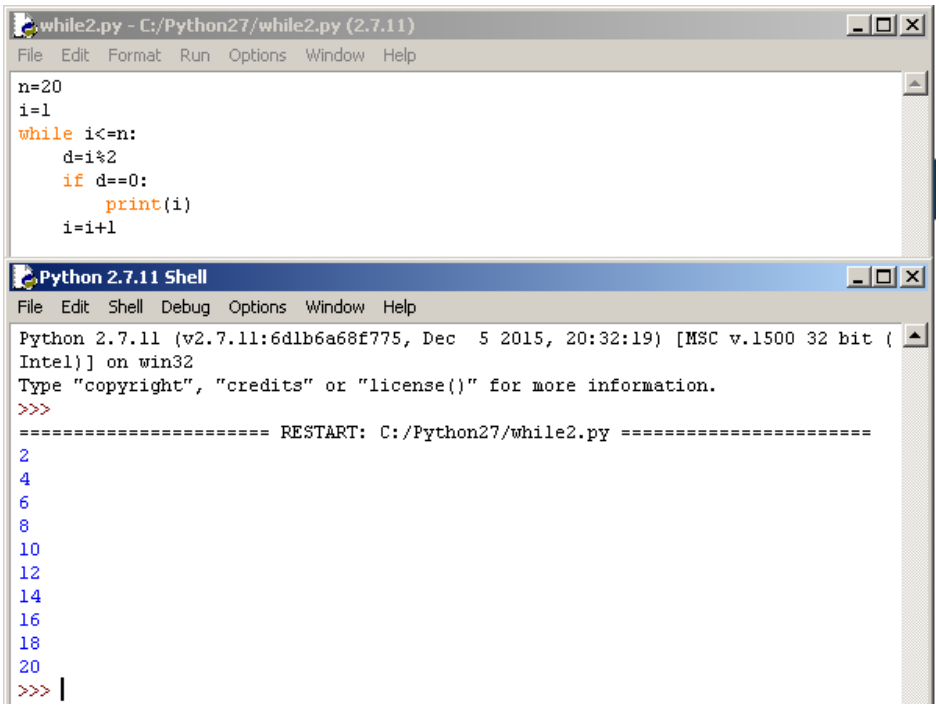
Вхідні дані	Відповідь
7051	1057 7510

Приклади для розв'язування завдання

1. Побудувати таблицю значень функції y = sin *x*, якщо значення аргументу змінюється від 0 до 100 з кроком 2.

```
import math
n = 100
k = 2
x = 0
while x <= n:
        y = math.sin(x)
        print (x, ' ' , y)
        x = x + k
```

2. Визначити парні числа у ряді чисел від 1 до 20



3. Алгоритм пошуку суми (добутку) елементів масиву

```
n=input('кількість елементів в масиві')
A=[]
s=0
for i in range(n-1):
    A.append(input('задайте',i,' елемент'))
    s=s+a[i]
print(s)
```

Розділ 5. Функції

Говорячи простими словами, функція є способом групування набору операторів, що дозволяє виконувати їх більше одного разу в програмі - упакованою процедурою, що викликається на ім'я. Функції здатні обчислювати результуюче значення і також дають можливість вказувати параметри, які є вхідними даними функції і можуть відрізнятися при кожному виконанні коду. Кодування операції як функції робить її загалом корисним інструментом, який можна використовувати у різноманітних контекстах.

По суті, функції пропонують альтернативу програмуванню шляхом вирізування та вставки — замість того, щоб мати безліч надлишкових копій коду операції, ми можемо винести його в єдину функцію. Тим самим ми радикально скорочуємо обсяг майбутньої роботи: якщо пізніше операцію потрібно модифікувати, то нам доведеться вносити зміни лише до одиночної копії коду функції, а не до численних копій, розкиданих по всій програмі.

Функції також є найбільш базовою програмною структурою Python, яка призначена для доведення до максимуму багаторазового використання коду і підводить нас до ширших понять проектування програм. Як буде показано, функції дозволяють розбивати складні системи на частини, що піддаються управлінню. За рахунок реалізації кожної частини у вигляді функції ми робимо її багаторазово застосовуваною та легкою для кодування.

5.1. Основи

Хоча й не дуже формально, але в попередніх розділах ми вже використовували кілька функцій. Скажімо, ми використовували вбудовану функцію len, щоб запросити кількість елементів в об'єкті колекції.

У цьому розділі ми дізнаємося, як писати нові функції в Python. Написані нами функції поводяться подібно до вбудованих функцій, які ми вже бачили: вони викликаються у виразах, набувають значення та повертають результати.

Але написання нових функцій вимагає докладання ряду додаткових ідей, які поки що не були представлені. Крім того, функції в Python поводяться зовсім не так, як у мовах, що компілюються на кшталт C.

Оператор def створює об'єкт функції та надає його імені. Загальний формат def виглядає так:

```
def ім'я(аргумент1, аргумент2, . . . аргументN) :
    оператори
```

Подібно до всіх складових операторів Python оператор def складається з рядка заголовка, за яким йде блок операторів, зазвичай з відступом (або одиночний оператор після двокрапки). Блок операторів стає тілом функції, тобто. кодом, який Python виконує щоразу, коли функція пізніше викликається.

У рядку заголовка def вказується ім'я функції, якому присвоюється об'єкт функції, а також список із нуля та більше аргументів (іноді званих параметрами) у круглих дужках. Іменам аргументів у заголовку надаються об'єкти, що передаються в круглих дужках у точці виклику.

Тіло функції часто містить оператор return:

```
def ім'я(аргумент1, аргумент2, . . . аргументN) :
    ...
    return значення
```

Оператор return у Python може з'являтися будь-де у тілі функції; після досягнення він закінчує виклик функції і посилає результат назад коду, що викликає. Оператор return складається з необов'язкового вираження з об'єктним значенням, що дає результат функції. Якщо значення опущено, то return відправляє назад None.

Оператор return сам собою також необов'язковий; якщо він відсутній, то вихід із функції відбувається, коли потік управління досягає кінця тіла функції. Формально функція без оператора return автоматично повертає об'єкт None, але таке значення, що повертається зазвичай при викликі ігнорується.

Оператор def в Python є справжнім оператором, що виконується: при виконанні він створює новий об'єкт функції і присвоює його імені. Оскільки

def — оператор, може з'являтися скрізь, де допускається оператор, навіть усередині інших операторів. Наприклад, хоча оператори def зазвичай виконуються, коли включає модуль імпортується, також цілком законно вкладати def всередину оператора if для вибору між альтернативними визначеннями функції:

```
if test:
     def func () :
          pass
else:
     def func () :
          pass
func ()
```

Один із способів зрозуміти цей код - усвідомити, що оператор def багато в чому схожий на оператор = він просто присвоює ім'я під час виконання. На відміну від компілюваних мов на кшталт функції Python не потребують повного визначення перед запуском програми. У більш загальному плані оператори def не оцінюються до тих пір, поки досягнуто і виконано, і код всередині операторів def не виконується до виклику функций.

Через те, що визначення функції відбувається під час виконання, в імені функції немає нічого особливого. Важливим є об'єкт, на який воно посилається:

```
othername = func
othername()
```

Нижче показано визначення функції на ім'я times, яке повертає добуток двох аргументів:

```
def times(x, y) :
     return x * y
```

Коли Python досягає і виконує цей оператор def, він створює новий об'єкт функції, що вміщає код функції, і присвоює його ім'я times. Зазвичай такий оператор знаходиться у файлі модуля і виконується під час його імпортування; проте для чогось невеликого цілком достатньо інтерактивної підказки.

Оператор def створює функцію, але не викликає її. Після виконання def функцію можна викликати (виконувати) у своїй програмі, додаючи до імені функції круглі дужки. Круглі дужки можуть додатково містити один і більше об'єктів-аргументів, що підлягають передачі (привласнення) імен у заголовку функції:

```
times (2, 4)
8
```

Вираз виклику передає в times два аргументи. Як згадувалося раніше, аргументи передаються по присвоюванню, так що у випадку імені x в заголовку функції присвоюється значення 2, у присвоюється значення 4 і тіло функції виконується. Тілом функції times є лише оператор return, який відправляє назад результат як значення виразу виклику. Об'єкт, що повертається, був виведений інтерактивно (як і в більшості мов, у Python значенням 2 * 4 буде 8), але якщо об'єкт необхідно використовувати пізніше, тоді його можна привласнити змінною. Ось приклад:

```
>>> x = times (3.14, 4)
>>> x
12.56
```

Давайте подивимося, що відбувається, коли функція викликається з передачею об'єктів різних видів:

```
>>> times (’Ni’, 4)
’NiNiNiNi'
```

Як тільки з'ясувалося, сам сенс виразу x * у нашої простої функції times залежить цілком від видів об'єктів, що вказуються для x і у; таким чином, та сама функція здатна виконувати множення в одній ситуації і повторення в іншій. Python залишає на розсуд об'єктів робити щось розумне з погляду синтаксису. Взагалі кажучи, операція * є лише координуючим механізмом, який передає контроль оброблюваним об'єктам.

Поведінка із залежністю від типів подібного роду відома як поліморфізм – термін, який по суті означає, що сенс операції залежить від об'єктів, які вона обробляє. Через те, що Python є мовою, що динамічно типізується,

поліморфізм в ній буквально процвітає. Фактично кожна операція в Python має поліморфізм: висновок, індексація, операція * і багато інших.

Така поведінка невипадково і великою мірою пояснює лаконічність та гнучкість мови. Наприклад, єдина функція зазвичай може застосовуватися до цілої категорії типів об'єктів автоматично. Доки об'єкти підтримують очікуваний інтерфейс (він же протокол), функція може обробляти їх. Тобто коли об'єкти, що передаються в функцію, мають очікувані методи та операції виразів, вони вважаються автоматично сумісними з логікою функції.

Навіть у нашій простій функції times це означає, що будь-які дра. об'єкти, які підтримують операцію * , будуть працювати незалежно від того, що вони являють собою і коли закодовані. Функція times буде працювати з двома числами (виконуючи множення), з рядком і числом (виконуючи повторення) або з будь-якою іншою комбінацією об'єктів, що підтримують очікуваний інтерфейс - навіть з об'єктами, заснованими на класах, які ми ще не придумали.

Більше того, якщо передані об'єкти нс підтримують очікуваний інтерфейс, Python виявить помилку при спробі виконати вираз * і автоматично згенерує виняток. Отже, зазвичай немає сенсу писати код перевірки щодо помилок. Насправді додавання такого коду перевірки обмежило корисність нашої функції, оскільки вона змогла б працювати тільки з об'єктами, чиї типи перевіряються.

Давайте розглянемо другий приклад функції, що робить щось корисніше, ніж перемноження аргументів, і додатково ілюструє основи функцій: накопичує елементи, спільні двох рядків.

До цього часу, ймовірно, ви вже здогадалися, що вихід із скрутного становища передбачає упаковку циклу for всередину функції. Таке рішення має кілька переваг.

• Розміщення коду у функцію робить його інструментом, який можна запускати стільки разів, скільки потрібно.

• Оскільки код, що викликає, може передавати довільні аргументи, функції досить універсальні для того, щоб працювати з будь-якими двома послідовностями (або іншими об'єктами, що ітеруються), перетин яких необхідно отримати.

• Коли логіка упакована всередину функції, у разі зміни реалізації перетину код доведеться модифікувати лише в одному місці.

• Поміщення коду функції у файл модуля означає, що її можна імпортувати та багаторазово застосовувати у будь-якій програмі, що виконується на комп'ютері.

Насправді приміщення коду на функцію перетворює її на універсальну утиліту перетину:

```
def intersect(seq1, seq2):
     res = []
     for x in seq1:
          if x in seq2:
               res.append(x)
     return res

>>> s1 = "SPAM"
>>> s2 = "SCAM"
>>> intersect(s1, s2)
['S', 'A', 'M']
```

Тут ми передаємо два рядки і отримуємо список, що містить загальні символи. Алгоритм, використовуваний функцією, простий: “кожного елемента у першому аргументі виконати перевірку, чи міститься він у другому аргументі, і якщо так, то додати їх у результат”. На мові Python алгоритм виглядає коротшим, ніж природною мовою, але зміст залишається тим же.

Як і всі добропорядні функції Python, функція intersect поліморфна, тобто. працює на довільних типах за умови, що вони підтримують очікуваний інтерфейс об'єкта:

```
>>> x = intersect ([1, 2, 3] , (1, 4))
>>> x
```

```
[1]
```

У цьому прикладі ми передали нашій функції об'єкти різних типів – список і кортеж (різнорідні типи) – і вона, як і раніше, обрала спільні елементи. Оскільки немає необхідності вказувати типи аргументів заздалегідь, функція intersect проходить по об'єктах послідовностей будь-якого виду, що передаються, поки вони підтримують очікувані інтерфейси.

5.2. Області видимості

Мабуть найцікавішою частиною розглянутого прикладу є його імена. Справа в тому, що змінна res всередині intersect є так званою локальною змінною - ім'я, яке видно тільки в коді всередині оператора def функції і існує лише на період виконання функції. Фактично через те, що всі імена, які тим чи іншим чином присвоюються всередині функції, за умовчанням класифікуються як локальні змінні, майже всі імена функції intersect виявляються локальними змінними:

• змінною res виробляється очевидне присвоєння, отже вона локальна;

• аргументи передаються з присвоєння, тому seql і seq2 також локальні;

• цикл for надає елементи змінної, а тому ім'я x — також локальна змінна.

Всі перелічені локальні змінні починають своє існування, коли функція викликається, і зникають при виході з функції - оператор return в кінці intersect посилає об'єкт, що результує, але ім'я res припиняє існувати. З цієї причини змінні функції не запам'ятовують значення між викликами; хоча повертається функцією об'єкт залишається, збереження інших видів інформації про стан потребує інших методик.

Все вже готове приступити до написання власних функцій, але вам необхідно отримати більш формальне уявлення про призначення імен у Python. У разі використання імені в програмі Python створює, змінює або шукає ім'я всередині того, що називається простір імен — місце, де імена існують. Коли йдеться про пошук значення імені стосовно коду, термін

область видимості належить до простору імен: тобто. розташування присвоювання імені у вихідному коді визначає область, де ім'я є видимим для коду.

Майже все, що пов'язане з іменами, включаючи класифікацію областей видимості, у Python відбувається під час присвоєння. Як ми вже бачили, імена в Python починають своє існування, коли їм вперше надаються значення, і щоб імена можна було використовувати, їм мають бути присвоєні значення. Оскільки імена заздалегідь не оголошуються, Python застосовує розташування присвоювання імені для асоціювання (тобто зв'язування) зі специфічним простором імен. Іншими словами, місце присвоювання імені у вихідному коді визначає простір імен, в якому воно існуватиме, і звідси його область видимості.

Крім пакування коду для багаторазового використання функції додають до програм додатковий рівень, щоб звести до мінімуму можливість виникнення конфліктів між змінними з одним і тим же ім'ям — за умовчанням усі імена, яким виконано присвоєння всередині функції, асоціюються з простором імен цієї функції та жодним іншим. . Ось що вказане правило означає.

• Імена, присвоєні всередині def, можна побачити лише в коді всередині цього оператора def. Посилатися на такі імена ззовні функції не можна.

• Імена, присвоєні всередині def, не конфліктують зі змінними за межами def, навіть якщо ті самі імена застосовуються десь в іншому місці. Ім'я X, присвоєне поза заданим оператором def (тобто в іншому def або на верхньому рівні файлу модуля), являє собою змінну, абсолютно відрізняється від імені X, присвоєну всередині цього def.

У всіх випадках область видимості змінної (де вона може використовуватися) завжди визначається місцем її присвоєння у вихідному коді і не має жодного відношення до того, яка функція викликає. Насправді змінні можуть присвоюватися в трьох різних місцях, що відповідають трьом різним областям видимості:

• якщо змінна присвоюється всередині def, вона буде локальної у цій функції;

• якщо змінна присвоюється в охоплюючому def, тоді вона буде нелокальною щодо вкладених функцій;

• якщо змінна присвоюється поза всіма def, вона буде глобальної загалом файле.

Ми називаємо це лексичною областю видимості, тому що області видимості змінних визначаються повністю розташуванням змінних у файлах вихідного коду програми, а не викликами функцій.

Наприклад, у наступному файлі модуля присвоєння X = 99 створює глобальну змінну на ім'я X (видиму в даному файлі), але присвоєння X = 88 створює локальну змінну X (видиму тільки всередині оператора def):

```
>>> X = 99
>>> def func():
... X = 88
func()
>>> X
99
```

Незважаючи на те, що обидві змінні мають ім'я X, пов'язані з ними області видимості роблять їх різними. Сукупний ефект полягає в тому, що області дії функцій допомагають уникнути конфліктів імен у програмах і зробити функції самодостатніми програмними одиницями — у їхньому коді не доведеться турбуватися про імена, що застосовуються в інших місцях програми.

Оператор global і споріднений з nonlocal з Python 3.X — єдині речі в Python, які віддалено нагадують оператори оголошення. Проте, вони не є оголошеннями типу чи розміру, а є оголошення просторів імен. Оператор global повідомляє Python, що функція планує змінювати одне і більше світових імен, тобто. імен, які існують в області видимості (просторі імен), що включає модуля.

Ось короткий огляд.

• Глобальні імена — це змінні, присвоєні на верхньому рівні модуля, що включає їх файл.

• Глобальні імена повинні оголошуватися, лише якщо вони виконують присвоєння всередині функції.

• На глобальні імена можна посилатися всередині функції без оголошення.

Іншими словами, global дозволяє змінювати імена, які існують поза def на верхньому рівні файлу модуля. Як буде показано пізніше, оператор nonlocal майже ідентичний, але застосовується до імен в локальній області видимості об'ємного оператора def, а не до імен у модулі, що включає.

Оператор global складається з ключового слова global, за яким слідує одне і більше імен, розділених комами. Привласнення або посилання на всі перераховані імена в тілі функції буде призводити до їх відображення на область видимості модуля, наприклад:

```
X = 88
def func():
     global X
     X = 99
func()
print(X)
99
```

Нижче показаний більш складний приклад використання global

```
y, z = 1, 2
def all_global():
     global x
     x = y + z
```

Змінні x, y та z є глобальними у функції all_global. Змінні y та z глобальні через те, що вони не присвоюються всередині функції; змінна x є глобальною тому, що вона була вказана в операторі global з метою явного відображення на область видимості модуля. Якби оператор global був відсутній, тоді завдяки присвоєнню змінна x вважалася б локальною.

5.3. Аргументи функцій

Передача аргументів – спосіб надсилання функцій об'єктів як вхідних даних. Аргументи (називаються також параметрами) присвоюються іменам функції, але вони мають більше відношення до посилань на об'єкти, ніж до областей видимості змінних. Python надає додаткові інструменти, такі як ключові аргументи, аргументи зі стандартними значеннями, а також збирачі та екстрактори довільних аргументів, які уможливлюють більшу гнучкість у способі передачі аргументів функції.

Аргументи передаються з присвоєння. У результаті виникає кілька наслідків, які не завжди очевидні новачкам, які будуть розкриті в цьому розділі. Нижче наведено короткий виклад ключових аспектів передачі аргументів функцій.

• Аргументи передаються шляхом автоматичного надання об'єктів іменам локальних змінних. Аргументи функцій, тобто. посилання на (можливо) об'єкти, що розділяються, відправлені викликаючим кодом, являють собою ще один приклад присвоєння Python в дії. Оскільки посилання реалізовані як покажчиків, всі аргументи насправді передаються через покажчики. Об'єкти, які передаються як аргументи, ніколи автоматично не копіюються.

• Присвоєння імен аргументів всередині функції не зачіпає код, що викликає. Коли функція виконується, імена аргументів у заголовку функції стають новими локальними іменами в області видимості функції. Ніякого суміщення імен аргументів функції та імен змінних в області видимості коду, що викликає, не відбувається.

• Модифікація всередині функції аргументу, що є об'єктом, що змінюється, може торкнутися код, що викликає. З іншого боку, оскільки аргументам просто присвоюються об'єкти, що передаються, у функціях можна модифікувати передані змінні об'єкти на місці, в результаті впливаючи на код, що викликає. Змінювані аргументи можуть бути вхідними і вихідними даними для функцій.

Щоб оцінити характеристики передачі аргументів у роботі, розглянемо наступний код:

```
>>> def f (a) :
          a = 99
>>> b = 88
>>> f (b)
>>> print (b)
88
```

У наведеному прикладі змінної a присвоюється об'єкт 88 в момент, коли функція викликається за допомогою f (Ь), але існує тільки всередині викликаної функції. Зміна всередині функції ніяк не впливає на місце, де функція викликалася; воно просто встановлює локальну змінену a в зовсім інший об'єкт.

Ось що означає відсутність суміщення імен - привласнення імені аргументу всередині функції (наприклад, a = 99) не змінює чарівним чином змінну на зразок b в області видимості, де викликано функцію. Імена аргументів можуть спочатку розділяти об'єкти, що передаються (по суті вони є вказівниками на ці об'єкти), але тільки тимчасово, коли функція викликається вперше. Після повторного надання імені аргументу зв'язок розривається.

Принаймні так відбувається у разі надання самим іменам аргументів. Коли аргументи передаються змінювані об'єкти, подібні до списків і словників, ми повинні усвідомлювати, що зміни на місці таких об'єктів можуть продовжити своє існування після завершення функції, а тому впливати на код, що викликає. Нижче наведено приклад, що демонструє описану поведінку:

```
>>> def changer (a, b) :
```

```
        a = 2
        b[0] = 'spam'
>>> X = 1
>>> L = [1, 2]
>>> changer (X, L)
>>> X, L
(1, ['spam', 2])
```

Функція changer надає значення самому аргументу і компоненту об'єкта, який посилається аргумент b. Ці два присвоєння всередині функції лише злегка відрізняються за синтаксисом, але призводять до різних результатів.

• Оскільки a — ім'я локальної змінної в області видимості функції, перше присвоєння не впливає на код, що викликає; воно просто модифікує локальну змінну a, щоб a посилалася на інший об'єкт, і не змінює прив'язку імені X в області видимості коду, що викликає. Ситуація така сама, як у попередньому прикладі.

• Аргумент b також є ім'ям локальної змінної, але йому переданий об'єкт, що змінюється (список, на який посилається L в області видимості коду, що викликає). Оскільки друге привласнення є зміна об'єкта дома, результат присвоювання b[0] у функції впливає значення L після повернення управління з функції.

Насправді другий оператор присвоювання в changer не змінює b - він модифікує частину об'єкта, який посилається b в даний момент. Така зміна на місці впливає тільки на код, що викликає, тому що модифікований об'єкт залишається існувати після виклику функції. Ім'я L теж не змінюється (воно, як і раніше, посилається на той самий модифікований об'єкт), але виглядає так, ніби ім'я L після виклику стало іншим, тому що значення, на яке воно посилається, було змінено всередині функції. Фактично список на ім'я L служить вхідними та вихідними даними для функції.

Ми вже обговорювали оператор return та застосовували його в кількох прикладах. Існує ще один спосіб використання цього оператора: через те, що

return здатний відправляти об'єкт будь-якого виду, він може повертати множинні значення, упаковуючи їх в кортеж або колекцію іншого типу. Насправді, хоча Python не підтримує те, що в деяких мовах називається передачею аргументів за посиланням, ми зазвичай можемо емулювати його, повертаючи кортежі та привласнюючи результати вихідним іменам аргументів у коді, що викликає:

```
>>> def multiple (x, y) :
          x = 2
          y = [3, 4]
          return x, y
>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple (X, L)
>>> X, L
(2, [3, 4])
```

Виглядає так, ніби код повертає два значення, але насправді значення лише одне — двоелементний кортеж, в якому опущені необов'язкові круглі дужки, що оточують. Після повернення керування з виклику можна застосувати привласнення кортежів для розпакування частин поверненого кортежу. Сукупний ефект такого кодового шаблону полягає в тому, що він відправляє множинні результати назад і емулює вихідні параметри в інших мовах за допомогою явних присвоювань. Тут X і L змінюються після виклику, але лише оскільки так реалізовано в коді.

Як було показано, аргументи Python завжди передаються по присвоювання; іменам в заголовку def надаються передані об'єкти. Проте, крім цієї моделі, Python пропонує додаткові інструменти, які змінюють спосіб зіставлення об'єктів-аргументів у виклику з іменами аргументів у заголовку до моменту присвоєння. Всі інструменти такого роду необов'язкові, але вони дозволяють писати функції, які підтримують гнучкіші шаблони виклику, до того ж, зустрічаються бібліотеки, які їх вимагають.

За умовчанням аргументи зіставляються за позицією, ліворуч, і необхідно передавати рівно стільки аргументів, скільки є імен аргументів у заголовку функції. Однак можна також задавати зіставлення на ім'я, надавати стандартні значення і використовувати збирачі для додаткових аргументів.

Якщо ви не використовуєте якийсь спеціальний синтаксис зіставлення, то Python зіставляє імена за ліворуч направо подібно до більшості інших мов. Наприклад, якщо ви визначили функцію, яка потребує трьох аргументів, тоді маєте викликати її з трьома аргументами:

```
>>> def f (a, b, c) : print (a, b, c)
>>> f (1, 2, 3)
1 2 3
```

Тим не менш, у Python ви можете бути більш точними щодо того, що відбувається під час виклику функції. Ключові аргументи уможливлюють зіставлення на ім'я, а не на позицію. Нижче використовується та сама функція:

```
>>> f(c=3, b=2, a=1)
1 2 3
```

Тут c=3 означає відправлення значення 3 аргументу на ім'я c. Більш формально Python зіставляє ім'я з у виклику з аргументом на ім'я з у заголовку визначення функції і потім передає цьому аргументу значення 3. Сукупний ефект даного виклику такий же, як у попереднього, але зверніть увагу, що коли застосовуються ключові слова, порядок дотримання аргументів зліва право несуттєвий, оскільки аргументи зіставляються на ім'я, а чи не по позиции. В одному виклику можна навіть комбінувати позиційні та ключові аргументи. У такому разі спочатку зіставляються всі позиційні аргументи зліва направо в заголовку, а потім ключові аргументи зіставляються на ім'я:

```
>>> f (1, c=3, b=2)
1 2 3
```

Стандартні значення згадувалися раніше під час обговорення областей видимості вкладених функцій. Коротко висловлюючись, стандартні значення дозволяють робити обрані аргументи функції необов'язковими; якщо значення

аргументу був передано, перед виконанням функції йому присвоюється стандартне значення.

Наприклад, ось функція з одним обов'язковим аргументом та двома аргументами, що мають стандартні значення:

```
>>> def f (a, b=2, c=3) : print(a, b, c)
```

При виклику такої функції ми маємо надати значення для a, чи з позиції, чи з ключовому слову; проте передача значень b і з необов'язкова. Якщо ми не передамо значення b та c, тоді вони отримають стандартні значення 2 та 3 відповідно:

```
>>> f (1)
1 2 3
>>> f (a=1)
1 2 3
>>> f (1, 4)
1 4 3
>>> f (1, 4, 5)
1 4 5
```

Ще два розширення при зіставленні * і ** призначені для підтримки функцій, які приймають будь-яку кількість аргументів. Обидва розширення можуть зустрічатися або у визначенні функції, або у виклику функції, і в зазначених двох місцях вони мають пов'язані цілі.

Коли розширення * застосовується у визначенні функції, воно забезпечує збирання непоставлених позиційних аргументів до кортежу:

```
>>> def f( * args):  print(args)
```

При виклику такої функції Python збирає всі позиційні аргументи в новий кортеж і надає його змінної args. Оскільки це нормальний об'єкт кортежу, він допускає індексацію, прохід у циклі for і т.д.

```
>>> f ()
()
>>> f (1)
(1)
```

```
>>> f (1, 2, 3, 4)
(1, 2, 3, 4)
```

Розширення ** схоже, але тільки працює з ключовими аргументами - воно збирає їх у новий словник, який можна обробляти за допомогою звичайних інструментів для словників. У певному сенсі форма ** дозволяє перетворювати ключові слова на словники, які потім можна проходити за допомогою викликів keys, словникових ітераторів і т.п. (приблизно так робить виклик dict при передачі йому ключових слів, але він повертає новий словник):

```
>>> def f( ** args)  : print (args)
>>> f()
{}
>>> f(a=l, b=2)
{'a' : 1, 'b': 2}
```

Нарешті, у заголовках функцій можна комбінувати нормальні аргументи з розширеннями * та **, щоб реалізовувати дуже гнучкі сигнатури викликів. Наприклад, у наступній взаємодії значення 1 передається за позицією, 2 і 3 збираються в кортеж pargs з позиційними аргументами, а x і y потрапляють у словник kargs з ключовими аргументами:

```
>>> def f(a, * pargs,  ** kargs) : print(a, pargs, kargs)
>>> f(l, 2, 3, x=l, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

У всіх останніх випусках Python ми можемо використовувати синтаксис * також під час виклику функції. У такому контексті цей синтаксис має сенс, протилежний його сенсу у визначенні функції, він розпаковує колекцію аргументів, а не збирає її. Наприклад, ми можемо передати чотири аргументи в кортежі і дозволити Python розпакувати їх в індивідуальні аргументи:

```
>>> def func (a, b, c, d) : print (a, b, c, d)
>>> args = (1, 2)
>>> args += (3, 4)
>>> func( * args)   # Те ж саме, що й func(l, 2, 3, 4)
1 2 3 4
```

Подібним чином синтаксис ** у виклику функції розпаковує словник пар ключ/значення в окремі ключові аргументи:

```
>>> args = {’a’: 1, 'b’: 2, ’c’: 3}
>>> args [ ’d’ ] = 4
>>> func( ** args) # Те ж саме, що й func(a=1, b=2, c=3, d=4)
1 2 3 4
```

Завдання для виконання

1. Створити функцію для обчислення значень функції y(x) на відрізку [a, b] з кроком h. Результат подати у вигляді таблиці, перший стовбець якої – значення аргумента, другий – відповідні значення функції.

2. Створити функцію обчислення площі трикутнику за трьома сторонами.

3. Створити програму для обчислення площі правильного шестикутника зі стороною a, використовуючи функцію обчислення площі трикутника з попереднього завдання.

4. Написати функцію для знаходження суми найбільшого та найменшого з трьох чисел.

5. Написати функцію знаходження факторіалу числа із використанням рекурсії.

6. Написати рекурсивну і без рекурсії функції знаходження k-го члену послідовності Фібоначчі.

7*. Створити програму для обчислення суми: 2! + 4! + 6! + ... + n! ($n \leq 16$, n - парне).

8*. Напишіть рекурсивну функцію для знаходження суми цифр цілого числа.

9 (просто або * або ** - залежить від оптимальності розв’язку). Створити функцію, яка визначає чи є її аргумент простим числом.

10. Створити функцію, яка виводить прості числа із заданого діапазону, використовуючи функцію попереднього завдання

11**. Створити функцію, яка виводить прості числа від 2 до N (введеного користувачем), використовуючи «решето Ератосфена». Знайти таке N, для якого ця функція стане краще за функцію попереднього завдання

12**. Написати рекурсивну функцію для переведення числа з десяткової системи числення у будь-яку іншу, задану користувачем, але основа системи числення менше за 10.

13**. Написати программу для гри “пятнашки”.

На квадратному полі розміром 4 на 4 з допомогою генератора випадкових чисел розставлено 15 фішок з номерами від 1 до 15.

11	1	12	7
10	14	5	9
3		2	13
8	4	15	6

Є одна вільна позиція. Розставить фішки за збільшенням їх номерів. Передвигати фішку можна лише на сусідню вільну позицію.

Приклади для розв’язування завдання

1. Приклад використання локальних та глобальних змінних

```
# x - глобальна змінна
x=5
def f():
    # x - локальна змінна
    x=10
    print (x)
    return
print (x)
```

2. Приклад використання оператору **global**

```
x=5 # x - глобальна змінна
def f():
    global x
    x=10        # x - глобальна змінна
    print (x)
    return
print (x)
```

3. Приклад обчислення суми ряду 1, 2, 3, …, n за допомогою рекурсії:

```
def sumr(n):
    if n==1:
        return 1
    return sumr(n-1)+n
```

Література

1. Путівник мовою програмування Python [Електронний ресурс]. – Режим доступу : http://pythonguide.rozh2sch.org.ua.
2. ArcGIS Pro Python [Електронний ресурс]. – Режим доступу : https://pro.arcgis.com/ru/pro-app/arcpy/main/arcgis-pro-arcpy-reference.htm.
3. Python. Обучение программированию [Электронный ресурс]. – Режим доступа : https://younglinux.info/python
4. Python ООП: https://python-scripts.com/object-oriented-programming-in-python#pros-cons-oop
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. А. Мюллер, С. Гвидо - Введение в машинное обучение с помощью Python. Руководство для специалистов по работе с данными – М. 2017. — 393 с.

14. Буйначев С.К., Боклаг Н.Ю. - Основы программирования на языке Python – Екатеренбург. - 2014. — 90 с.

15. Бэрри П. - Изучаем программирование на Python (Мировой компьютерный бестселлер). - М.- 2017. — 618 с.

16. Зед Шоу - Легкий способ выучить Python (Мировой компьютерный бестселлер) - 2017. — 353 с.

17. Федоров Д. - Основы программирования на примере языка Python - М.- 2018. – 167 с.

18. Чан Уэсли Дж. - Python. создание приложений (Библиотека профессионала) –М. – 2015.- 794 с.