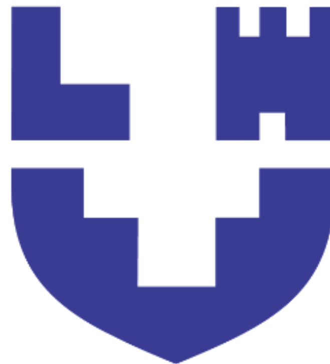
Міністерство освіти і науки України
Луцький національний технічний університет



ПРОГРАМУВАННЯ НА PYTHON

Конспект лекцій
для здобувачів першого (бакалаврського) рівня вищої освіти
галузі знань 12 (F) Інформаційні технології
денної та заочної форм навчання

Луцьк 2026

УДК 004.432 (07)
П 78

Рекомендовано до видання вченою радою факультету КІТ ЛНТУ,
протокол № ______ від « _____ » _________ 20 26 року.

Голова вченої ради факультету КІТ ________________ Інна КОНДІУС

Електронна копія друкованого видання передана для внесення в репозитарій ЛНТУ

Директор бібліотеки ________________ Наталія ПОЛІЩУК

Розглянуто і схвалено на засіданні кафедри комп’ютерної інженерії та безпеки ЛНТУ, протокол № 9 від « _____ » 04 20 26 року.

Укладачі: ________________ Сергій КОСТЮЧКО, кандидат технічних наук, доцент кафедри комп’ютерної інженерії та безпеки ЛНТУ

________________ Людмила КОНКЕВИЧ, асистент кафедри комп’ютерної інженерії та безпеки, ЛНТУ

Рецензент: ________________ Микола ПОЛІЩУК, кандидат технічних наук, доцент кафедри комп’ютерної інженерії та безпеки ЛНТУ

Відповідальний за випуск: ________________ Тарас ТЕРЛЕЦЬКИЙ, кандидат технічних наук, доцент кафедри комп’ютерної інженерії та безпеки ЛНТУ

П 78 Програмування на Python. Конспект лекцій для здобувачів першого (бакалаврського) рівня вищої освіти галузі знань 12 (F) Інформаційні технології денної та заочної форм навчання / уклад. С.М. Костючко, Л.М. Конкевич. Луцьк: ЛНТУ, 2026. 81 с.

Конспект лекцій призначений для здобувачів першого (бакалаврського) рівня вищої освіти галузі знань 12 (F) Інформаційні технології.

ВСТУП

ТЕМА 1. ВСТУП 5
1.1 Обчислювальне моделювання 5
1.2 Чому Python використовують для наукових обчислень? 8
1.3 Версія Python 12
ТЕМА 2. ПОТУЖНИЙ КАЛЬКУЛЯТОР 14
2.1 Підказка Python та цикл читання-виведення-друку (REPL) 14
2.2 Калькулятор 15
2.3 Цілочисельне ділення 16
2.4 Математичні функції 18
2.5 Змінні 19
2.6 Неможливі рівняння 21
ТЕМА 3. ТИПИ ДАНИХ ТА СТРУКТУРА ДАНИХ 23
3.1 Який це тип? 23
3.2 Числа 23
3.3 Послідовності 26
3.4 Передача аргументів функціям 38
ТЕМА 4. СПОСТЕРЕЖЕННЯ 47
4.1 dir() 47
4.2 type 49
4.3 isinstance 49
ТЕМА 5. INPUT AND OUTPUT 50
5.1 Друк на стандартному виводі (зазвичай на екрані) 50
5.2 Читання та запис файлів 55
ТЕМА 6. КОНТРОЛЬ ПОТОКУ 59
6.1 Основи 59
6.2 If-then-else 61
6.3 Цикл for 62
6.4 Цикл while 62

6.5 Реляційні оператори (порівняння) у операторах if і while 63
6.6 Винятки 64
ТЕМА 7. ФУНКЦІЇ ТА МОДУЛІ 68
7.1 Вступ 68
7.2 Використання функцій 68
7.4 Значення за замовчуванням та необов’язкові параметри 73
7.5 Модулі 73
ПЕРЕЛІК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ 80

ТЕМА 1. ВСТУП

1.1 Обчислювальне моделювання

1.1.1 Вступ

В наш час багато процесів та систем досліджуються або розробляються за допомогою комп'ютерного моделювання. Нові прототипи літальних апаратів, такі як для недавнього A380, спочатку розробляються та перевіряються практично за допомогою комп'ютерного моделювання. З огляду на постійно зростаючу обчислювальну потужність, доступну через суперкомп'ютери, кластери комп'ютерів і навіть настільні та портативні машини, ця тенденція, ймовірно, буде продовжуватися. Комп'ютерне моделювання зазвичай використовується у фундаментальних дослідженнях, щоб допомогти зрозуміти експериментальні вимірювання та замінити, наприклад, зростання та виготовлення дорогих зразків / експериментів, де це можливо. У промисловому контексті проектування виробів та пристроїв часто може бути здійснено набагато ефективніше, якщо здійснюється практично за допомогою моделювання, а не за допомогою побудови та випробування прототипів. Це особливо стосується районів, де зразки є дорогими, такі як нанонаука (де дорого створювати дрібниці) та аерокосмічна промисловість (де дорого будувати великі речі). Бувають також ситуації, коли певні експерименти можна проводити лише віртуально (починаючи від астрофізики і вивчаючи наслідки масштабних ядерних або хімічних аварій). Обчислювальне моделювання, включаючи використання обчислювальних інструментів для подальшої обробки, аналізу та візуалізації даних, застосовується в техніці, фізиці та хімії протягом багатьох десятиліть, але стає все більш важливим через дешеву доступність обчислювальних ресурсів. Обчислювальне моделювання також починає відігравати більш важливу роль у вивченні біологічних систем, економіки, археології, медицини, охорони здоров'я та багатьох інших областей.

1.1.2 Обчислювальне моделювання

Для вивчення процесу за допомогою комп’ютерного моделювання ми виділяємо два етапи. Перший - це розробка моделі реальної системи. Вивчаючи рух невеликого предмета, наприклад, копійки, скажімо, під впливом сили тяжіння, ми можемо ігнорувати тертя повітря: наша модель може враховувати лише силу тяжіння та інерцію копійки, тобто $a(t) = \frac{F}{m} = -\frac{9.81м}{c^2}$ – це наближення реальної системи. Модель, як правило, дозволить нам виразити поведінку системи (в деякій наближеній формі) за допомогою математичних рівнянь, які часто включають звичайні диференціальні рівняння (ЗДР) або часткові диференціальні екватони (ЧДР). У таких природничих науках, як фізика, хімія та супутня інженерія, часто, не так складно знайти відповідну модель, але отримані рівняння, як правило, дуже важко розв’язати, і в більшості випадків рівняння взагалі не можуть бути вирішені аналітично. З іншого боку, у суб’єктів, які не так добре описані через математичні рамки і залежать від поведінки об’єктів, дії яких неможливо передбачити детерміновано (наприклад, людей), набагато складніше знайти хорошу модель для опису реальності. Як правило, в цих дисциплінах отримані рівняння легше розв’язувати, але їх важче знайти, а обгрунтованість моделі більше підпадає під сумнів. Типовими прикладами є спроби імітувати економіку, використання глобальних ресурсів, поведінку панічної натовпу і т.д. Вище описано розробку моделей для опису реальності, але використання цих моделей не обов’язково залучає будь-які комп’ютери або чисельну роботу. Насправді, якщо рівняння моделі можна розв’язати аналітично, тоді слід це зробити і записати розв’язки рівняння. На практиці навряд чи будь-які модельні рівняння систем, що представляють інтерес, можуть бути розв’язані аналітично, і саме тут потрібен комп’ютер: використовуючи чисельні методи, ми можемо принаймні вивчити модель для певного набору граничних умов. У розглянутому вище прикладі ми не можемо легко побачити з числового рішення, що швидкість копійки під впливом сили тяжіння змінюватиметься лінійно з часом (що ми можемо

легко побачити з аналітичного розв’язку, доступного для цієї простої системи: $v(t) = t \cdot \frac{9.81м}{c^2} + v_0$. Чисельне рішення, яке можна обчислити за допомогою комп’ютера, буде складатися з даних, які показують, як швидкість змінюється з часом для певної початкової швидкості v_0 (v_0 тут є граничною умовою). Комп’ютерна програма повідомляє довгі списки двох чисел, що зберігають (i) значення часу t_i, для якого було обчислено певне (ii) значення швидкості v_i. Побудувавши графік усіх v_i навпроти t_i, або побудувавши криву на основі даних, ми можемо зрозуміти залежність даних (що ми можемо просто побачити з аналітичного рішення, звичайно). Потрібно знаходити аналітичні рішення, але кількість проблем, де це можливо зробити – невелика. Зазвичай отримання числового результату комп’ютерного моделювання є дуже корисним (незважаючи на недоліки чисельних результатів порівняно з аналітичним виразом), оскільки це єдиний можливий спосіб вивчення системи взагалі. Назва обчислювального моделювання випливає з двох етапів: (I) моделювання, тобто знаходження опису моделі реальної системи, та (II) вирішення отриманих рівнянь моделі за допомогою обчислювальних методів, оскільки це єдиний спосіб, яким рівняння взагалі можна розв’язати.

1.1.3 Програмування для підтримки обчислювального моделювання

Існує велика кількість пакетів, що забезпечують можливості обчислювального моделювання. Якщо вони задовольняють потреби досліджень або проектування, а будь-яка обробка даних та візуалізація належним чином підтримується за допомогою існуючих інструментів, можна проводити дослідження обчислювального моделювання без будь-яких глибших знань програмування. В дослідницькому середовищі, як в академічних колах, так і при дослідженні нових продуктів / ідей / ... у промисловості, часто досягається така точка, коли існуючі пакети не зможуть виконати необхідне завдання моделювання, або де більше можна дізнатися, проаналізувавши існуючі дані новинними способами тощо. На цей момент потрібні навички програмування. Також загалом корисно мати широке

розуміння будівельних блоків програмного забезпечення та основних ідей програмної інженерії, оскільки ми використовуємо все більше і більше пристроїв із програмним контролем. Часто забувають, що комп'ютер не може зробити нічого, чого не можемо зробити ми –люди. Однак комп'ютер може зробити щось набагато швидше, а також робить набагато менше помилок. Таким чином, в обчисленнях, які виконує комп'ютер, немає магії: їх могли робити люди, і – насправді – це було багато років (див., наприклад, запис у Вікіпедії про Людський комп'ютер).

Розуміння того, як побудувати комп'ютерне моделювання, зводиться приблизно до: (I) пошуку моделі (часто це означає пошук правильних рівнянь), (II) знання, як розв'язувати ці рівняння чисельно, (III) реалізації методів для обчислення цих рішень (це біт програмування).

1.2 Чому Python використовують для наукових обчислень?

Основна увага при розробці мови Python зосереджена на продуктивності та читабельності коду, наприклад:

– інтерактивна консоль (оболонка) python;

– дуже чіткий, читабельний синтаксис;

– сильні можливості самоаналізу;

– повна модульність, підтримка ієрархічних пакетів;

– обробка помилок на основі винятків;

– динамічні типи даних та автоматичне управління пам'яттю.

Оскільки Python є інтерпретованою мовою, і вона працює в рази повільніше, ніж скомпільований код, можна запитати, чому хтось повинен розглядати таку «повільну» мову для комп'ютерного моделювання? На цю критику є дві відповіді:

1. Час реалізації проти часу виконання. Не лише час виконання сприяє вартості обчислювального проекту – також потрібно враховувати вартість робіт з розробки та технічного обслуговування. У перші дні наукових

обчислень (скажімо, в 1960/70/80) обчислювальний час був настільки дорогим, що було цілком розумно інвестувати багато місяців часу програміста, щоб покращити ефективність обчислення на кілька відсотків. Однак сьогодні цикли процесора стали набагато дешевшими, ніж час програміста. Для дослідницьких кодів, які часто виконуються лише невелику кількість разів (до того, як дослідники перейдуть до наступної проблеми), може бути економічно визнати, що код працює лише на 25% від очікуваної можливої швидкості, якщо це заощаджує, скажімо, місяць часу дослідника (або програмістів). Наприклад: якщо час виконання фрагмента коду становить 10 годин, і можна передбачити, що він буде виконуватися приблизно 100 разів, то загальний час виконання приблизно 1000 годин. Було б чудово, якби це можна було зменшити до 25% і заощадити 750 (CPU) годин. З іншого боку, чи варто додаткове очікування (близько місяця) та вартість 750 годин процесора інвестувати в один місяць часу людини (хтось міг би зробити щось інше, поки триває обчислення)? Часто відповідь не така. Зчитування та обслуговування коду: короткий код – менше помилок. Пов’язана проблема полягає в тому, що дослідницький код використовується не лише для одного проекту, а продовжують використовувати його знову і знову, розвивається, росте, роздвоюється тощо. У цьому випадку часто виправдано вкладати більше часу, щоб зробити код швидким. У той же час значна частина часу програміста піде на (I) внесення необхідних змін, (II) тестування їх ще до того, як почнеться робота з оптимізації швидкості зміненої версії. Щоб мати можливість підтримувати, розширювати та модифікувати код, часто непередбачуваними способами, може бути корисним використання мови, яка читається та має велику виразну силу.

2. Добре написаний код Python може бути дуже швидким, якщо критичні за часом частини виконуються за допомогою компільованої мови. Як правило, менше 5% відсотків кодової бази проекту моделювання потребує більше 95% часу виконання. Поки ці розрахунки виконуються дуже ефективно, не потрібно турбуватися про всі інші частини коду, оскільки

загальний час, який займає їх виконання, є незначним. Інтенсивна обчислювальна частина програми повинна бути налаштована на досягнення оптимальної продуктивності. Python пропонує ряд опцій.

Наприклад, розширення numpy Python надає інтерфейс Python до складених та ефективних бібліотек LAPACK, які є квазістандартними в числовій лінійній алгебрі. Якщо досліджувані проблеми можуть бути сформульовані таким чином, що з часом великі системи алгебраїчних рівнянь повинні бути вирішені або обчислені власні значення тощо, тоді може бути використаний скомпільований код у бібліотеці LAPACK (через пакет Python-numpy). На цьому етапі обчислення проводяться з такою ж продуктивністю як і Fortran/C, оскільки, по суті, використовується код Fortran/C. Matlab, до речі, використовує саме це: мова сценаріїв (скриптів) Matlab працює дуже повільно (приблизно в 10 разів повільніше, ніж Python), але Matlab працює швидше шляхом делегування операції matrix до скомпільованих бібліотек LAPACK.

Існуючі числові бібліотеки C/Fortran можна взаємодіяти, щоб використовувати їх усередині Python (використовуючи, наприклад, Swig, Boost.Python і Cython).

Python може бути розширений за допомогою скомпільованих мов, якщо вимоглива до обчислень частина задачі є алгоритмічно нестандартною і не можна використовувати існуючі бібліотеки. Часто використовуються C, Fortran та C ++ для реалізації швидких розширень.

Нижче перелічено деякі інструменти, які використовуються для використання скомпільованого коду з Python. Розширення scipy.weave корисно, якщо в C потрібно виразити лише короткий вираз. Інтерфейс Cython набуває популярності, щоб напівавтоматично оголошувати типи змінних у коді Python, перекладати цей код на C (автоматично), а потім використовувати скомпільований код C з Python. Cython також використовується для швидкого обгортання існуючої бібліотеки C інтерфейсом, щоб бібліотеку C можна було використовувати з Python.

Boost.Python спеціалізується на обтіканні коду C ++ у Python. Висновок полягає в тому, що Python «досить швидкий» для більшості обчислювальних завдань, і що його зручна мова високого рівня часто компенсує знижену швидкість у порівнянні зі скомпільованими мовами нижчого рівня. Поєднання Python із спеціально складеним скомпільованим кодом для продуктивності критичних частин коду в більшості випадків дає практично оптимальну швидкість.

1.2.1 Стратегії оптимізації

Ми, як правило, розуміємо скорочення часу виконання, обговорюючи «оптимізацію коду» в контексті обчислювального моделювання, і по суті, ми хочемо виконати необхідні обчислення якомога швидше. (Іноді нам потрібно зменшити обсяг оперативної пам’яті, обсяг виводу даних на диск або в мережу.) У той же час нам потрібно переконатися, що ми не витрачаємо невідповідну кількість часу програмування для досягнення такої швидкості: як завжди, повинен бути баланс між часом програмістів та покращенням, яке ми можемо отримати від цього.

1.2.2 Спочатку все правильно, а потім швидко

Щоб ефективно писати швидкий код у правильному порядку потрібно: (I) спочатку написати програму, яка здійснює правильний розрахунок. Для цього вибрати мову/підхід, який дозволяє швидко писати код і змусити його швидко працювати незалежно від швидкості виконання. Потім (II) або змініть програму, або перепишіть її з нуля на тій самій мові, щоб пришвидшити виконання. Під час процесу продовжуйте порівнювати результати із повільною версією, написаною першою, щоб переконатися, що оптимізація не призводить до помилок. (Ознайомившись із поняттям регресійних тестів, їх слід використовувати тут для порівняння нового, і, сподіваємось, більш швидкого коду з оригінальним кодом.) Типовим шаблоном у Python є початок написання чистого коду Python, а потім використання бібліотек Python, які використовувати скомпільований код внутрішньо (наприклад, швидкі масиви, які надає Numpy, та процедури від

Scipy, які повертаються до встановлених числових кодів, таких як ODEPACK, LAPACK та інші). Якщо потрібно, то можна після ретельного профілювання почати замінювати частини коду Python компільованою мовою, такою як C та Fortran, щоб додатково покращити швидкість виконання (як обговорювалося вище).

1.2.3 Прототипування на Python

Виявляється, якщо певний код повинен бути написаний, скажімо, на C++, прототипувати код на Python (часто) ефективніше в часі, і як тільки буде знайдено відповідний дизайн (і структуру класу), перекладіть код на C ++.

1.3 Версія Python

Існує дві версії мови Python: Python 2.x та Python 3.x. Вони (дещо) відрізняються - зміни в Python 3.x були введені для усунення недоліків у конструкції мови, які були виявлені з моменту створення Python. Прийнято рішення про прийняття певної несумісності для досягнення вищої мети - кращої мови на майбутнє. Для наукових обчислень вирішальним є використання числових бібліотек, таких як numpy, scipy та пакет побудови matplotlib. Всі вони доступні для Python 2.x, і все частіше вони також доступні для Python 3 (насправді всі бібліотеки, перелічені на сьогодні, перенесені). Оскільки Python 2.x як і раніше є Python за замовчуванням у багатьох системах, і існує чимала кількість дослідницьких кодів на основі Python 2, ми будемо використовувати Python 2.x у цій книзі. Однак ми напишемо код, максимально можливий у стилі Python 3 (і зрозумілий Python 2). Найвидатнішим прикладом є те, що в Python 2.x команда print є особливою, де, як і в Python 3, це звичайна функція. Наприклад, у Python 2.7 ми можемо написати

```
print " Hello World "
```

де, як і в Python 3, це призведе до помилки SyntaxError. Правильним способом використання друку в Python 3 буде функція

```
print ( " Hello World " )
```

На щастя, позначення функції (в дужках) також дозволено в Python 2.7, тому ми вибираємо це позначення в наших прикладах, і таким чином вони будуть виконуватися в Python 2.7 і Python 3.x. (Є й інші відмінності.) Перехід усіх активно підтримуваних кодів з Python 2 на Python 3, ймовірно, займе принаймні ще 5 років, а може і 10. Може статися, що Python 2.7 буде ще довго активно використовуватись.

ТЕМА 2. ПОТУЖНИЙ КАЛЬКУЛЯТОР

2.1 Підказка Python та цикл читання-виведення-друку (REPL)

Python – це інтерпретована мова. Ми можемо або збирати послідовності команд у текстові файли, і зберігати їх у файлі як програму Python. Загальновизнано, що ці файли мають розширення файлу .py, наприклад hello.py. Ми також можемо вводити окремі команди в підказці Python, які негайно обчислюються та виконуються інтерпретатором Python. Це дуже корисно для програміста / учня, щоб зрозуміти, як користуватись певними командами (часто перед тим, як скласти ці команди у довшу програму Python). Роль Python можна описати як читання команди, її оцінку, друк оцінюваного значення та повторення циклу (цикл) - це походження абревіатури REPL.

Щоб запустити інтерпретатор, ми можемо:

1. У вікнах: запустити IDLE.

2. У вікнах: знайти підказку MS-DOS і ввести python.exe, а потім ключ повернення.

3. У Linux / Unix / Mac OSX: знайдіть оболонку (у програмах / утилітах на Mac OS X називається «термінал» і введіть python, а потім клавішу повернення. Запит python (chevron >>>) сигналізує про те, що Python чекає введення від нас:

```
>>>
```

Тепер ми можемо вводити команди, наприклад 4 + 5, а потім клавішу RETURN.

```
>>> 4+5
9
>>>
```

Після того, як ми натиснемо клавішу повернення, Python оцінить вираз (4+ 5) і відобразить обчислюване значення (9) у наступному рядку. Потім він відображає підказку python (>>>) у наступному рядку, щоб вказати, що він готовий до наступного введення. Це інтерактивне середовище програмування

іноді називають циклом читання-друку (REPL), оскільки вираз читається, обчислюється, результат друкується, а потім цикл починається знову.

2.2 Калькулятор

Базові операції, такі як додавання (+), віднімання (-), множення (*), ділення (/) та піднесення до степеня (**) працюють (здебільшого), як очікувалося:

```
>>> 10+10000
10010
>>> 42 -1.5
40.5
>>> 47*11
517
>>> 10/0.5
20.0
>>> 2**2
4
>>> 2**3
8
>>> 2**4
16
>>> 2+2
4
>>> # Це коментар
... 2+2
4
>>> 2+2 # і коментар у тому ж рядку, що і код
4
```

використовуючи той факт, що $\sqrt[n]{x} = x^{\frac{1}{n}}$, ми можемо обчислити$\sqrt{3} = 1.732050\ldots$ за допомогою **:

```
>>> 3**0.5
1.7320508075688772
```

Дужки можна використовувати для групування:

```
>>> 2*10+5
25
>>> 2*(10+5)
30
```

2.3 Цілочисельне ділення

Неочікувана поведінка може статися при діленні двох цілих чисел:

```
>>> 15/6
2
```

Це явище відоме (у багатьох мовах програмування, включаючи C) як цілочисельне ділення: оскільки ми надаємо оператору ділення (/) два цілих числа (15 і 6), припущення, яке робить Python, полягає в тому, що ми шукаємо значення числа цілого типу. Математично правильна відповідь (число з плаваючою точкою) 2.5. Цілочисельне ділення полягає у скороченні дробових цифр та запису лише цілочисельної частини (тобто 2 у цьому прикладі). Його також називають «ділення з остачею».

2.3.1 Як уникнути цілочисельного ділення?

Існує два способи уникнути проблеми цілочисельного ділення:

1. Скористайтеся майбутнім поділом Python: було вирішено, що починаючи з Python 3.0 і далі оператор ділення повертає число з плаваючою комою (комплексні, якщо потрібно), навіть якщо чисельник та знаменник цілого типу. Цю функцію можна активувати у старих (2.x) версіях Python за допомогою функції from future import division:

```
>>> 21/7
3
>>> 15/6
2
>>> from __future__ import division
>>> 15/6
2.5
>>> 21/7
3.0
```

Якщо потрібно використати цю функцію в програмі python, то, зазвичай, вона міститься на початку файлу.

2. Як варіант, якщо ми переконаємось, що принаймні одне число (чисельник або знаменник) має тип числа з плаваючою точкою (або комплексне), результатом оператора ділення буде число з плаваючою точкою. Це можна зробити, написавши 15. замість 15, змусивши перетворити число на плаваюче:

```
>>> 15/6
2
>>> 15./6
2.5
>>> 15.0/6
2.5
>>> float (15)/6
2.5
>>> 15/6.
2.5
>>> 15/ float (6)
2.5
>>> 15./6.
2.5
```

Для цілочисельного ділення, ми можемо використовувати //: результатом 1//2 є 0 (використовується у версіях 2.x, 3.x та в найближчому майбутньому).

2.3.2 Чому потрібно звернути увагу на проблему ділення?

Цілочисельне ділення може призвести до помилок: припустимо, ви пишете код для обчислення середнього значення m=(x+y)/2 для двох чисел x та y. Можна записати це

```
m = ( x + y ) / 2
```

Припустимо, що x=0.5, y=0.5, тоді матимемо m=0.5 (тому що 0.5+0.5=1.0, тобто 1.0 - це число з плаваючою комою, і, отже, 1.0/2=0.5). Або,

x=10, y=30, і, оскільки, 10+30 =40 і 40/2=20, ми отримуємо правильну відповідь m=20. Однак, якби взяти цілі числа x=0 і y=1, тоді отримаємо m=0 (оскільки 0+1=1 і 1/2 обчислюється як 0), тоді як m=0.5 правильна відповідь. Можемо змінити наведений вище рядок коду для правильної його роботи. Ось три версії:

```
m = ( x + y ) / 2.0
```

```
m = float ( x + y ) / 2
```

```
m = ( x + y ) * 0.5
```

Цілочисельне ділення є поширеним серед більшості мов програмування (включаючи важливі C, C ++ та Fortran), і важливо знати про цю проблему.

2.4 Математичні функції

Оскільки Python – це мова програмування загального призначення, загальновживані математичні функції, такі як sin, cos, exp, log та багато інших, знаходяться в математичному модулі з іменем math. Ми можемо скористатися цим, як тільки імпортуємо математичний модуль:

```
>>> import math
>>> math . exp (1.0)
2.7182818284590451
```

За допомогою функції dir ми можемо побачити каталог об’єктів, доступних в математичному модулі:

```
>>> dir( math )
[ ‘ __doc__ ‘ , ‘ __file__ ‘ , ‘ __name__ ‘ , ‘ acos ‘ , ‘ asin ‘ , ‘ atan ‘ , ‘ atan2 ‘ ,
‘ ceil ‘ , ‘ cos ‘ , ‘ cosh ‘ , ‘ degrees ‘ , ‘e ‘ , ‘ exp ‘ , ‘ fabs ‘ , ‘ floor ‘ ,
‘ fmod ‘ , ‘ frexp ‘ , ‘ hypot ‘ , ‘ ldexp ‘ , ‘ log ‘ , ‘ log10 ‘ , ‘ modf ‘ , ‘ pi ‘ ,
‘ pow ‘ , ‘ radians ‘ , ‘ sin ‘ , ‘ sinh ‘ , ‘ sqrt ‘ , ‘ tan ‘ , ‘ tanh ‘]
```

Функція довідки може надати більше інформації про модуль (help(math)) щодо окремих об’єктів:

```
>>> help ( math . exp )
Help on built -in function exp in module math :
```

```
exp (...)
exp ( x )
Return e raised to the power of x .
```

Математичний модуль визначає для констант π та e значення:

```
>>> math . pi
3.1415926535897931
>>> math . e
2.7182818284590451
>>> math . cos ( math . pi )
-1.0
>>> math . log ( math . e )
1.0
```

2.5 Змінні

Змінна може використовуватися для зберігання певного значення або об’єкта. У Python усі числа (і все інше, включаючи функції, модулі та файли) є об’єктами. Змінна створюється через присвоєння:

```
>>> x = 0.5
>>>
```

Як тільки змінна x буде створена шляхом присвоєння 0.5 у цьому прикладі, ми можемо використовувати її:

```
>>> x *3
1.5
>>> x **2
0.25
>>> y = 111
>>> y +222
333
```

Змінна замінюється, якщо призначено нове значення:

```
>>> y = 0.7
>>> math . sin ( y ) ** 2 + math . cos ( y ) ** 2
1.0
```

Знак рівності (‘=‘) використовується для присвоєння значення змінній. Після цього результат не відображається перед наступним інтерактивним запитом:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Значення може бути присвоєно декільком змінним одночасно:

```
>>> x = y = z = 0 # initialise x , y and z with 0
>>> x
0
>>> y
0
>>> z
0
```

Перш ніж їх можна використовувати, потрібно створити змінні (присвоїти їм значення), або виникне помилка:

```
>>> # try to access an undefined variable
... n
Traceback ( most recent call last ):
File " < stdin > " , line 1 , in < module >
NameError : name ‘n ‘ is not defined
```

В інтерактивному режимі змінній присвоюється останній надрукований вираз. Це означає, що коли ви використовуєте Python як настільний калькулятор, дещо простіше продовжувати обчислення, наприклад:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
```

2.5.1 Термінологія

Строго кажучи, наступне трапляється, коли ми пишемо

```
>>> x = 0.5
```

Спочатку Python створює об’єкт 0.5. Усе в Python є об’єктом, як і число з плаваючою комою 0.5. Цей об’єкт зберігається десь у пам’яті. Далі Python прив’язує ім’я до об’єкта. Ім’я є x, і ми часто випадково посилаємося до x як

змінної, об’єкту або навіть значенням 0.5. Однак технічно x - це ім’я, яке прив’язане до об’єкта 0.5. Інший спосіб сказати це - x є посиланням на об’єкт. Хоча часто достатньо подумати про присвоєння 0.5 змінній x, існують ситуації, коли нам потрібно пам’ятати, що насправді відбувається. Зокрема, коли ми передаємо посилання на об’єкти функціям, нам слід усвідомити, що функція може діяти на об’єкті (а не на копії об’єкта).

2.6 Неможливі рівняння

У комп’ютерних програмах ми часто знаходимо такі твердження, як

```
x = x + 1
```

Якщо ми читаємо це як рівняння, як ми звикли з математики

$$x = x + 1$$

ми можемо відняти x з обох сторін, щоб знайти, що

$$0 = 1$$

Ми знаємо це неправда, тому тут щось не так. Відповідь полягає в тому, що «рівняння» в комп’ютерних кодах - це не рівняння, а завдання. Їх завжди потрібно читати наступним чином двоступеневим способом:

1. Обчислити значення з правого боку знака рівності.

2. Призначити це значення імені змінної, яке показано зліва. (У Pyton: прив’яжіть ім’я ліворуч до об’єкта, показаного праворуч.)

Інколи в літературі використовують такі позначення для вираження завдань та уникнення плутанини з математичними рівняннями:

$$x \leftarrow x + 1$$

Давайте застосуємо наше двоступеневе правило до присвоєння x = x + 1, наведеного вище:

1. Оцініть значення праворуч від знака рівності: для цього нам потрібно знати, яке поточне значення x. Припустимо, що x на даний момент є 4. У такому випадку права сторона x+1 оцінюється як 5.

2. Призначте це значення (тобто 5) імені змінної, яке відображається зліва від x

```
>>> x = 4
>>> x = x + 1
>>> print x
5
```

2.6.1 Позначення + =

Оскільки це досить поширена операція збільшення змінної x на якусь фіксовану величину c, ми можемо записати

```
x += c
```

Замість

```
x = x + c
```

Наш початковий приклад вище, таким чином, міг бути написаний

```
>>> x = 4
>>> x += 1
>>> print x
5
```

Ті самі оператори визначені для множення з константою (* =), віднімання константи (- =) та ділення на константу (/ =). Зверніть увагу, що порядок + та = має значення

```
x + = 1
```

збільшить змінну x на одиницю, де як

```
x = + 1
```

буде присвоювати значення +1 змінній x

ТЕМА 3. ТИПИ ДАНИХ ТА СТРУКТУРА ДАНИХ

3.1 Який це тип?

Python знає різні типи даних. Щоб знайти тип змінної, використовуйте функцію type ():

```
>>> a = 45
>>> type ( a )
<type ‘ int ‘ >
>>> b = ‘ This is a string ‘
>>> type ( b )
<type ‘ str ‘ >
>>> c = 2 + 1 j
>>> type ( c )
<type ‘ complex ‘ >
>>> d = [1 , 3 , 56]
>>> type ( d )
<type ‘ list ‘ >
```

3.2 Числа

Посилання на бібліотеку Python: офіційний огляд числових типів, http://docs.python.org/library/stdtypes.html#numeric-types-int-float-long-complex

Вбудованими числовими типами є цілі числа (див. Розділ 3.2.1) та числа з плаваючою точкою (див. розділ 3.2.3) та складні числа з плаваючою комою (розділ 3.2.4). Існують також так звані довгі цілі числа (3.2.2, які не мають верхньої або нижньої межі (припускаючи, що машина забезпечує достатньо оперативної пам’яті)

3.2.1 Цілі числа

Ми бачили використання цілих чисел вже в розділі 2.2. Майте на увазі проблеми цілочисельного ділення (розділ 2.3). Якщо нам потрібно перетворити рядок, що містить ціле число, у ціле, ми можемо використовувати функцію int ():

```
>>> a = ‘ 34 ‘ # a is a string containing the characters 3 and 4
>>> x = int( a ) # x is in integer number
```

Функція int () також перетворює числа з плаваючою комою в цілі числа:

```
>>> int (7.0)
7
>>> int (7.9)
7
```

Зауважте, що int скорочує будь-яку нецілу частину числа з плаваючою комою. Щоб округлити число з плаваючою комою до цілого числа, скористайтеся командою round (), а потім перетворіть округлений float у int:

```
>>> round (7.9)
8.0
>>> int( round (7.9))
8
```

3.2.2 Довгі цілі числа

Нарівні з іншими мовами програмування та підтримкою цілочисельної арифметики за допомогою сучасних процесорів, існує верхня межа для цілих чисел, яку можна представити. У Python модуль sys надає такий номер у sys.maxint:

```
>>> import sys
>>> sys . maxint
2147483647
```

Якщо цей діапазон перевищений, то Python змінить тип числа з int на long:

```
>>> type ( sys . maxint )
<type ‘ int ‘ >
>>> sys . maxint +1
2147483648 L
>>> type ( sys . maxint +1)
<type ‘ long ‘ >
```

Тип довгого цілого числа поводиться як ціле число, але будь-яка арифметика, що включає довгі цілі числа, виконується на рівні програмного

забезпечення. Це дозволяє уникнути переповнення цілих чисел, але слід зазначити, що операції із довгими цілими числами є значно повільнішими, ніж операції з цілими числами. Не існує обмежень на максимальне або мінімальне довге ціле число, ніж могло б бути використано (хоча чим довше число, тим більше потрібно оперативної пам'яті і тим більше часу процесора для проведення обчислень). Немає порівнянного типу long int у мові C або Matlab. (У Python 3.0 різниця між int та long int зникне.)

3.2.3 Числа з плаваючою комою

Рядок, що містить число з плаваючою точкою, може бути перетворений у число з плаваючою комою за допомогою команди float()

```
>>> a = ' 35.342 '
>>> b = float ( a )
>>> print b
35.342
>>> print type ( b )
<type ' float ' >
```

3.2.4 Комплексні числа

Python (як Fortran та Matlab) має вбудовані комплексні числа. Ось кілька прикладів того, як їх використовувати:

```
>>> x = 1 + 3 j
>>> x
(1+3 j )
>>> abs( x ) # computes the absolute value
3.1622776601683795
>>> x . imag
3.0
>>> x . real
1.0
>>> x * x
( -8+6 j )
>>> x * x . conjugate ()
(10+0 j )
>>> 3 * x
```

```
(3+9 j )
```

Зверніть увагу, що якщо ви хочете виконувати більш складні операції (наприклад, квадратний корінь тощо), вам слід використовувати модуль cmath (Complex MATHematics):

```
>>> import cmath
>>> cmath . sqrt ( x )
(1.442615274452683+1.0397782600555705 j )
```

3.2.5 Функції, що застосовуються до всіх типів чисел

Функція abs () повертає абсолютне значення числа (також зване модулем):

```
>>> a = -45.463
>>> print abs ( a )
45.463
```

Зверніть увагу, що abs () також працює для комплексних чисел (див. 3.2.4)

3.3 Послідовності

Рядки (3.3.1), списки (3.3.2) і кортежі (3.3.3) є послідовностями. Їх можна проіндексувати (3.3.4) та нарізати (3.3.5) однаково. Кортежі та рядки є «незмінними» (що в основному означає, що ми не можемо змінювати окремі елементи всередині кортежу, і ми не можемо змінювати окремі символи всередині рядка), тоді як списки «змінюються». (Тобто ми можемо змінювати елементи у списку). Послідовності поділяють такі операції

a[i]	повертає i-й елемент
a[i:j]	повертає елементи i до j - 1
len(a)	повертає кількість елементів у послідовності
min(a)	повертає найменше значення в послідовності
max(a)	повертає найбільше значення у послідовності
x in a	повертає True якщо x є елементлм a

a + b	об’єднує a і b
n * a	створює n копій послідовності a

3.3.1 Тип послідовності 1: Рядок

Рядок – це (незмінна) послідовність символів. Рядок можна визначити, використовуючи одинарні лапки:

```
>>> a = ‘ Hello World ‘
```

подвійні лапки:

```
>>> a = " Hello World "
```

або потрійні лапки будь-якого виду

```
>>> a = """ Hello World """
>>> a = ‘‘‘ Hello World ‘‘‘
```

Тип рядка - str, а порожній рядок задається через "":

```
>>> a = " Hello World "
>>> type ( a )
<type ‘ str ‘ >
>>> b = " "
>>> type ( b )
<type ‘ str ‘ >
>>> type ( " Hello World " )
<type ‘ str ‘ >
>>> type ( " " )
<type ‘ str ‘ >
```

Кількість символів у рядку (тобто його довжина) можна отримати за допомогою функції len () - :

```
>>> a = " Hello Moon "
>>> len( a )
10
>>> a = ‘ test ‘
>>> len( a )
4
>>> len( ‘ another test ‘)
12
```

Ви можете комбінувати («об'єднати») два рядки за допомогою оператора +:

```
>>> ' Hello ' + ' World '
' Hello World '
```

Рядки мають ряд корисних методів, включаючи, наприклад, upper (), який повертає рядок у верхньому регістрі:

```
>>> a = " This is a test sentence . "
>>> a . upper ()
' THIS IS A TEST SENTENCE . '
```

Список доступних рядкових методів можна знайти в довідковій документації Python. Якщо доступне підказка Python, для отримання цієї інформації слід використовувати функцію dir і help, тобто dir ("") надає перелік методів, довідку можна використовувати для вивчення кожного методу. Особливо корисним методом є split (), який перетворює рядок у список рядків:

```
>>> a = " This is a test sentence . "
>>> a . split ()
[ ' This ' , ' is ' , 'a ' , ' test ' , ' sentence . ']
```

Метод split () відокремить рядок там, де знайде пробіл. Пробіл означає будь-який символ, який друкується як пробіл, наприклад, один пробіл або кілька пробілів або вкладку. Передаючи розділовий символ методу split (), рядок може розділитися на різні частини. Припустимо, наприклад, ми хотіли б отримати список повних речень

```
>>> a = " The dog is hungry . The cat is bored . The snake is awake . "
>>> a . split ( " . " )
[ ' The dog is hungry ' , ' The cat is bored ' , ' The snake is awake ' , ' ']
```

Протилежний рядовий метод split - join, який можна використовувати наступним чином:

```
>>> a = " The dog is hungry . The cat is bored . The snake is awake . "
>>> s = a . split ( '. ')
>>> s
[ ' The dog is hungry ' , ' The cat is bored ' , ' The snake is awake ' , ' ']
>>> " . " . join ( s )
```

```
‘ The dog is hungry . The cat is bored . The snake is awake . ‘
>>> " STOP " . join ( s )
‘ The dog is hungry STOP The cat is bored STOP The snake is awake STOP ‘
```

3.3.2 Тип послідовності 2: Список

Список - це послідовність об’єктів. Об’єкти можуть бути будь-якого типу, наприклад цілі числа:

```
>>> a = [34 , 12 , 54]
```

Порожній список представлений []:

```
>>> type ( a )
<type ‘ list ‘ >
>>> type ([])
<type ‘ list ‘ >
```

Як і у рядків, кількість елементів у списку може бути отримана за допомогою функції len ():

```
>>> a = [ ‘ dog ‘ , ‘ cat ‘ , ‘ mouse ‘]
>>> len( a )
3
```

Також можна змішувати різні типи в одному списку:

```
>>> a = [123 , ‘ duck ‘ , -42 , 17 , 0 , ‘ elephant ‘]
```

У Python список є об’єктом. Тому список може містити інші списки (оскільки список зберігає послідовність об’єктів):

```
a = [1 , 4 , 56 , [5 , 3 , 1] , 300 , 400]
```

Ви можете об’єднати ("об’єднати") два списки за допомогою оператора +:

```
>>> [3 , 4 , 5] + [34 , 35 , 100]
[3 , 4 , 5 , 34 , 35 , 100]
```

Або ви можете додати один об’єкт у кінець список за допомогою методу append ():

```
>>> a = [34 , 56 , 23]
>>> a . append (42)
>>> print a
[34 , 56 , 23 , 42]
```

Ви можете видалити об’єкт зі списку, викликавши метод remove () і передавши об’єкт для видалення. Наприклад:

```
>>> a = [34 , 56 , 23 , 42]
>>> a . remove (56)
>>> print a
[34 , 23 , 42]
```

Часто потрібен спеціальний тип списку (часто разом із циклами for-for), і для цього існує команда для створення цього списку: range (n) команда генерує список цілих чисел, починаючи з 0 і вгору, але не включаючи n. Ось кілька прикладів:

```
>>> range (3)
[0 , 1 , 2]
>>> range (10)
[0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9]
```

Ця команда часто використовується з для циклів for. Наприклад, для друку цифр 02, 12, 22, 32,. . . , 102, можна використовувати таку програму:

```
>>> for i in range (11):
... print i ** 2
...
0
1
4
9
16
25
36
49
64
81
100
```

Команда range приймає додатковий параметр для початку цілочисельної послідовності (старт) та іншого необов’язкового параметра для розміру кроку. Це часто записується як діапазон range([start],stop,[step]), де аргументи у квадратних дужках (тобто старт і крок) необов’язкові. Ось кілька прикладів:

```
>>> range (3 , 10) # start =3
[3 , 4 , 5 , 6 , 7 , 8 , 9]
```

```
>>> range (3 , 10 , 2) # start =3 , step =2
[3 , 5 , 7 , 9]
>>> range (10 , 0 , -1) # start =10 , step = -1
[10 , 9 , 8 , 7 , 6 , 5 , 4 , 3 , 2 , 1]
```

3.3.3 Тип послідовності 3: Кортежі

Кортеж - це (незмінна) послідовність об’єктів. Кортежі дуже схожі за поведінкою на списки, за винятком того, що вони не можуть бути змінені (тобто незмінні). Наприклад, об’єкти в послідовності можуть бути будь-якого типу:

```
>>> a = (12 , 13 , ‘ dog ‘)
>>> a
(12 , 13 , ‘ dog ‘)
>>> a [0]
12
```

Дужки не є необхідними для визначення кортежу: послідовність об’єктів, розділених комами, достатньо для визначення кортежу:

```
>>> a = 100 , 200 , ‘ duck ‘
>>> a
(100 , 200 , ‘ duck ‘)
```

хороша практика включати круглі дужки там, де це допомагає показати, що кортеж визначений. Кортежі також можна використовувати для одночасного виконання двох завдань:

```
>>> x , y = 10 , 20
>>> x
10
>>> y
20
```

Це можна використовувати для обміну об’єктами в одному рядку. Наприклад

```
>>> x = 1
>>> y = 2
>>> x , y = y , x
>>> print x
2
```

```
>>> print y
1
```

Порожній кортеж задано ()

```
>>> t = ()
>>> len( t )
0
>>> type ( t )
<type ‘ tuple ‘ >
```

Позначення кортежу, що містить одне значення, спочатку може здатися трохи дивним:

```
>>> t = (42 ,)
>>> type ( t )
<type ‘ tuple ‘ >
>>> len( t )
1
```

Додаткова кома потрібна, щоб відрізнити (42,) від (42), де в останньому випадку дужки читатимуться як визначальний пріоритет оператора: (42) спрощується до 42, що є просто числом:

```
>>> t = (42)
>>> type ( t )
<type ‘ int ‘ >
```

Цей приклад показує незмінність кортежу:

```
>>> a = (12 , 13 , ‘ dog ‘)
>>> a [0]
12
>>> a [0] = 1
Traceback ( most recent call last ):
File " < stdin > " , line 1 , in ?
TypeError : object doesn ‘t support item assignment
```

Незмінність є основною відмінністю кортежу від списку (останній можна змінювати). Ми повинні використовувати кортежі, коли не хочемо, щоб зміст змінювався. Зверніть увагу, що функції Python, які повертають більше одного значення, повертають їх кортежами (що має сенс, оскільки ви не хочете, щоб ці значення змінювались).

3.3.4 Індексація послідовностей

Доступ до окремих об’єктів у списках можна отримати за допомогою індексу об’єкта та квадратних дужок [ ]:

```
>>> a = [ ‘ dog ‘ , ‘ cat ‘ , ‘ mouse ‘]
>>> a [0]
‘ dog ‘
>>> a [1]
‘ cat ‘
>>> a [2]
‘ mouse ‘
```

Зверніть увагу, що Python (як C, але на відміну від Fortran та на відміну від Matlab) починає рахувати індекси з нуля! Python пропонує зручний ярлик для отримання останнього елемента у списку: для цього використовується індекс “-1”, де мінус вказує, що це один елемент із зворотного боку списку. Аналогічно індекс “-2” поверне другий останній елемент:

```
>>> a = [ ‘ dog ‘ , ‘ cat ‘ , ‘ mouse ‘]
>>> a [ -1]
‘ mouse ‘
>>> a [ -2]
‘ cat ‘
```

Якщо хочете, то можете думати, що індекс a [-1] є скороченим позначенням для [len (a) - 1]. Пам’ятайте, що рядки (наприклад, списки) також є типом послідовності, і їх можна індексувати однаково:

```
>>> a = " Hello World ! "
>>> a [0]
‘H ‘
>>> a [1]
‘e ‘
>>> a [10]
‘d ‘
>>> a [ -1]
‘! ‘
>>> a [ -2]
‘d ‘
```

3.3.5 Послідовності нарізки

Нарізання послідовностей може бути використано для отримання більш ніж одного елемента. Наприклад:

```
>>> a = " Hello World ! "
>>> a [0:3]
‘ Hel ‘
```

Пишучи [0: 3], ми запитуємо перші 3 елементи, починаючи з елемента 0. Аналогічним чином:

```
>>> a [1:4]
‘ ell ‘
>>> a [0:2]
‘ He ‘
>>> a [0:6]
‘ Hello ‘
```

Ми можемо використовувати негативні індекси для посилання на кінець послідовності:

```
>>> a [0: -1]
‘ Hello World ‘
```

Також можна пропустити початковий або кінцевий індекс, і це поверне всі елементи до початку або кінця послідовності. Ось кілька прикладів, щоб пояснити це:

```
>>> a = " Hello World ! "
>>> a [:5]
‘ Hello ‘
>>> a [5:]
‘ World ! ‘
>>> a [ -2:]
‘d ! ‘
>>> a [:]
‘ Hello World ! ‘
```

Зауважте, що [:] створить копію файлу. Деякі люди користуються індексами для нарізки як протилежні інтуїтивні. Найкращий спосіб згадати, як працюють зрізи, - це думати про індекси як про вказівку між символами, з

лівим краєм першого символу, пронумерованого 0. Тоді правий край останнього символу рядка з 5 символів має індекс 5, наприклад:

```
 | H  | e  |  l  |  l  |  o |
  + - - -+ - - -+ - - -+ - - -+ - - -+
  0      1      2      3      4      5   <-- use for SLICING
 -5    -4     -3     -2     -1          <-- use for SLICING
                                          from the end
```

Перший рядок чисел дає положення індексів нарізки 0 ... 5 у рядку; другий рядок дає відповідні негативні показники. Зріз від i до j складається з усіх символів між ребрами, позначеними i та j, відповідно. Отже, важливим твердженням є те, що для нарізки нам слід думати про індекси, що вказують між символами. Для індексації краще думати про індекси, що стосуються символів. Ось невеликий графік, що узагальнює ці правила:

```
     0      1      2       3     4       <-- use for INDEXING
    -5    -4     -3      -2    -1       <-- use for INDEXING
  + - - -+ - - -+ - - -+ - - -+ - - -+   from the end
  | H  | e  |  l  |  l  |  o |
  + - - -+ - - -+ - - -+ - - -+ - - -+
  0      1      2      3      4      5   <-- use for SLICING
 -5    -4     -3     -2     -1          <-- use for SLICING
                                          from the end
```

Якщо ви не впевнені, що це правильний індекс, це завжди хороша техніка, щоб пограти з маленьким прикладом у підказці Python, щоб перевірити речі до або під час написання програми.

3.3.6 Словники

Словники також називають «асоціативними масивами» та «хеш-таблицями». Словники – це невпорядковані набори пар ключ-значення. Порожній словник можна створити за допомогою фігурних дужок:

```
>>> d = {}
```

Пари ключового слова-значення можна додати так:

```
>>> d [ ‘ today ‘] = ‘ 22 deg C ‘            # ‘ today ‘ is the keyword
                                                # ‘22 deg C ‘ is the value
>>> d [ ‘ yesterday ‘] = ‘ 19 deg C ‘
```

d.keys () повертає список усіх ключів:

```
>>> d . keys ()
[ ‘ yesterday ‘ , ‘ today ‘]
```

Ми можемо отримати значення, використовуючи ключове слово як індекс:

```
>>> print d [ ‘ today ‘]
22 deg C
```

Інші способи заповнення словника, якщо дані відомі при створенні час:

```
>>> d2 = {2:4 , 3:9 , 4:16 , 5:25}
>>> d2
{2: 4 , 3: 9 , 4: 16 , 5: 25}
>>> d3 = dict ( a =1 , b =2 , c =3)
>>> d3
{ ‘a ‘: 1 , ‘c ‘: 3 , ‘b ‘: 2}
```

Функція dict () створює порожній словник. Інші корисні методи словника включають values(), items(), has_key():

```
>>> d . values ()
[ ‘ 19 deg C ‘ , ‘ 22 deg C ‘]
>>> d . items ()
[( ‘ yesterday ‘ , ‘ 19 deg C ‘) , ( ‘ today ‘ , ‘ 22 deg C ‘ )]
>>> d . has_key ( ‘ today ‘)
True
>>> d . has_key ( ‘ tomorrow ‘)
False
>>> d . get ( ‘ today ‘ , ‘ unknown ‘)
‘ 22 deg C ‘
>>> d . get ( ‘ tomorrow ‘ , ‘ unknown ‘)
‘ unknown ‘
>>> d . has_key ( ‘ today ‘)
True
>>> d . has_key ( ‘ tomorrow ‘)
False
>>> ‘ today ‘ in d                          # as d . haskey ( ‘ today ‘)
True
>>> ‘ tomorrow ‘ in d                       # as d . haskey ( ‘ tomorrow ‘)
False
```

Метод get(key,default) надасть значення для даного ключа, якщо цей ключ існує, інакше він поверне об'єкт за замовчуванням. Ось більш складний приклад:

```
order = {}                          # create empty dictionary
# add orders as they come in
order [ ' Peter '] = ' Pint of bitter '
order [ ' Paul '] = ' Half pint of Hoegarden '
order [ ' Mary '] = ' Gin Tonic '
# deliver order at bar
for person in order . keys ():
    print person , " requests " , order [ person ]
```

яка видає цей результат:

```
Paul requests Half pint of Hoegarden
Peter requests Pint of bitter
Mary requests Gin Tonic
```

Інші технічні характеристики:

1. Ключове слово може бути будь-яким (незмінним) об'єктом Python. Це включає: числа, струни, кортежі.

2. Словники дуже швидко отримують значення (коли надається ключ) Інший приклад, щоб продемонструвати перевагу використання словників над парами списків:

```
dic = {}                                          # create empty dictionary

dic [ " Hans " ] = " room 1033 "                  # fill dictionary
dic [ " Andy C " ] = " room 1031 "                # " Andy C " is key
dic [ " Ken " ] = " room 1027 "                   # " room 1027" is value

for key in dic . keys ():
    print key , " works in " , dic [ key ]
```

Результат:

```
Hans works in room 1033
Andy C works in room 1031
Ken works in room 1027
```

Без словника:

```
people = [ " Hans " ," Andy C " ," Ken " ]
rooms = [ " room 1033 " ," room 1031 " ," room 1027 " ]

# possible inconsistency here since we have two lists
if not len( people ) == len ( rooms ):
    raise RunTimeError , " people and rooms differ in length "
for i in range ( len( rooms ) ):
    print people [ i ] , " works in " , rooms [ i ]
```

3.4 Передача аргументів функціям

Python завжди передає (значення) посилання на об’єкт функції. Фактично це викликає функцію за посиланням, хоча можна було б позначати її як виклик за значенням (посилання). Ми розглядаємо аргументи, передані за значенням та посиланням, перш ніж обговорювати ситуацію в Python більш детально.

3.4.1 Виклик за значенням

Можна очікувати, що якщо ми передамо об’єкт за значенням функції, зміни цього значення всередині функції не вплинуть на об’єкт (оскільки ми передаємо не сам об’єкт, а лише його значення, яке є копією). Ось приклад такої поведінки (в С):

```
# include < stdio .h >
void pass_by_value (int m ) {
    printf ( " in pass_by_value : received m =% d \ n " ,m );
    m =42;
    printf ( " in pass_by_value : changed to m =% d \ n " ,m );
}
int main ( void ) {
  int global_m = 1;
  printf ( " global_m =% d \ n " , global_m );
  pass_by_value ( global_m );
  printf ( " global_m =% d \ n " , global_m );
  return 0;
}
```

разом з відповідним результатом:

```
global_m =1
in pass_by_value : received m =1
in pass_by_value : changed to m =42
global_m =1
```

Значення 1 глобальної змінної global_m не змінюється, коли функція передає значення змінює свій вхідний аргумент на 42.

3.4.2 Виклик за посиланням

З іншого боку, виклик функції за допомогою посилання означає, що об’єкт, який надається функції, є посиланням на об’єкт. Це означає, що функція побачить той самий об’єкт, що і у коді виклику (оскільки вони посилаються на один і той же об’єкт: ми можемо розглядати посилання як вказівник на місце в пам’яті, де знаходиться об’єкт). Будь-які зміни, що діють на об’єкт усередині функції, тоді будуть видимі в об’єкті на рівні виклику (оскільки функція насправді працює на одному об’єкті, а не на його копії). Ось один приклад, що показує це за допомогою покажчиків у C:

```
# include < stdio .h >
void pass_by_reference (int * m ) {
    printf ( " in pass_by_reference : received m =% d \ n " ,* m );
    * m =42;
    printf ( " in pass_by_reference : changed to m =% d \ n " ,* m );
}
int main ( void ) {
    int global_m = 1;
    printf ( " global_m =% d \ n " , global_m );
    pass_by_reference (& global_m );
    printf ( " global_m =% d \ n " , global_m );
    return 0;
}
```

разом із відповідним результатом:

```
global_m =1
in pass_by_reference : received m =1
in pass_by_reference : changed to m =42
global_m =42
```

C ++ надає можливість передавати аргументи як посилання, додаючи амперсанд перед іменем аргументу у функції визначення:

```
# include < stdio .h >

void pass_by_reference (int & m ) {
    printf ( " in pass_by_reference : received m =% d \ n " ,m );
    m =42;
    printf ( " in pass_by_reference : changed to m =% d \ n " ,m );
}

int main ( void ) {
    int global_m = 1;
    printf ( " global_m =% d \ n " , global_m );
    pass_by_reference ( global_m );
    printf ( " global_m =% d \ n " , global_m );
    return 0;
}
```

разом із відповідним результатом:

```
global_m =1
in pass_by_reference : received m =1
in pass_by_reference : changed to m =42
global_m =42
```

3.4.3 Передача аргументу в Python

У Python об’єкти передаються як значення посилання (вказівника) на об’єкт. Залежно за способом використання посилання у функції та залежно від типу об’єкта, на який посилається, це може призвести до поведінки передачі-посилання (де будь-які зміни об’єкта, отримані як функція аргумент, відразу відображаються на рівні виклику). Ось три приклади для обговорення цього. Ми починаємо з передачі списку функції, яка повторюється через всі елементи в послідовності і подвоює значення кожного елемента:

```
def double_the_values ( l ):
      print " in double_the_values : l = % s " % l
```

```
        for i in range (len( l )):
        l [ i ] = l [ i ] * 2
        print " in double_the_values : changed l to l = % s " % l
l_global = [0 , 1 , 2 , 3 , 10]
print ( " In main : s =% s " % l_global )
double_the_values ( l_global )
print ( " In main : s =% s " % l_global )
```

який видає цей результат:

```
In main : s =[0 , 1 , 2 , 3 , 10]
in double_the_values : l = [0 , 1 , 2 , 3 , 10]
in double_the_values : changed l to l = [0 , 2 , 4 , 6 , 20]
In main : s =[0 , 2 , 4 , 6 , 20]
```

Змінна l є посиланням на об’єкт списку. Рядок l [i] = l [i] * 2 спочатку обчислює праву частину і читає елемент з індексом i, а потім множить це на два. Потім посилання на цей новий об’єкт зберігається в об’єкті списку l у позиції з індексом i. Таким чином, ми змінили об’єкт списку, на який посилається через l. Посилання на об’єкт списку ніколи не змінюється: рядок

```
l [ i ] = l [ i ] * 2
```

змінює елементи l [i] списку l, але ніколи не змінює посилання l для списку. Таким чином, як функція, так і рівень виклику працюють на одному об’єкті за допомогою посилань l та global_l відповідно. Навпаки, ось приклад, коли не змінюють елементи списку в межах функції:

```
def double_the_list ( l ):
    print " in double_the_list : l = % s " % l
    l = l + l
    print " in double_the_list : changed l to l = % s " % l

l_global = " Hello "
print ( " In main : l =% s " % l_global )
double_the_list ( l_global )
print ( " In main : l =% s " % l_global )
```

який видає цей результат:

```
In main : l = Hello
in double_the_list : l = Hello
in double_the_list : changed l to l = HelloHello
```

```
In main : l = Hello
```

Тут відбувається те, що під час оцінки l = l + l створюється новий об'єкт, який вміщує l + l, і що ми потім прив'язуємо до нього ім'я l. У процесі ми втрачаємо посилання на об'єкт списку l, який був наданий функції (і, отже, ми не змінюємо об'єкт списку, наданий функції). Нарешті, давайте розглянемо

```
def double_the_value ( l ):
       print " in double_the_value : l = % s " % l
      l = 2 * l
     print " in double_the_values : changed l to l = % s " % l

l_global = 42
print ( " In main : s =% s " % l_global )
double_the_value ( l_global )
print ( " In main : s =% s " % l_global )
```

який видає цей результат:

```
In main : s =42
in double_the_value : l = 42
in double_the_values : changed l to l = 84
In main : s =42
```

У цьому прикладі ми також подвоюємо значення (з 42 до 84) у межах функції. Однак, коли ми прив'язуємо об'єкт 84 до імені python l (тобто рядка l = l * 2), ми створюємо новий об'єкт (84), і ми прив'язуємо новий об'єкт до l. У процесі ми втрачаємо посилання на об'єкт 42 у межах функції. Це не впливає ні на сам об'єкт 42, ні на глобальне посилання на нього.

Підсумовуючи, поведінка Python при передачі аргументів функції може здаватися різною. Тим не менш, це завжди виклик за значенням, де значення є посиланням на даний об'єкт, а поведінку можна пояснити за допомогою тих же міркувань в кожному конкретному випадку.

3.4.4 Міркування щодо продуктивності

Виклик функції за значенням вимагає копіювання значення перед передачею функції. З точки зору продуктивності (як часу виконання, так і вимог до пам'яті), це може бути вартісним процесом, якщо значення велике.

(Уявіть, що значення - це об’єкт numpy.array, який може мати розмір у кілька мегабайт або гігабайт.) Як правило, один користувач віддає перевагу виклику за посиланням для великих об’єктів даних, оскільки в цьому випадку передається лише вказівник на об’єкти даних, незалежно від фактичного розміру об’єкта, і, отже, це, як правило, швидше, ніж виклик за значенням. Таким чином, підхід Python до (ефективного) виклику за допомогою посилання є ефективним. Однак нам слід бути обережними, щоб наша функція не змінювала дані, які вони отримували там, де це небажано.

3.4.5 Ненавмисне внесення змін до даних

Як правило, функція не повинна змінювати дані, подані як вхідні дані до неї. Наприклад, наступний код демонструє спробу визначити максимальне значення списку і - ненароком - змінює список у процесі:

```
def mymax ( s ): # demonstrating side effect
    if len( s ) == 0:
        raise ValueError ( ‘ mymax () arg is an empty sequence ‘)
    elif len ( s ) == 1:
        return s [0]
    else :
        for i in range (1 , len( s )):
            if s [ i ] < s [ i - 1]:
                s [ i ] = s [ i - 1]
        return s [ len( s ) - 1]

s = [ -45 , 3 , 6 , 2 , -1]
print ( " in main before caling mymax ( s ): s =% s " % s )
print ( " mymax ( s )=% s " % mymax ( s ))
print ( " in main after calling mymax ( s ): s =% s " % s )
```

і видає це вивести в main перед викликом

```
in main before caling mymax ( s ): s =[ -45 , 3 , 6 , 2 , -1]
mymax ( s )=6
in main after calling mymax ( s ): s =[ -45 , 3 , 6 , 6 , 6]
```

Користувач функції mymax () не очікує, що вхідний аргумент буде змінено під час виконання функції. Як правило, нам слід уникати цього. Є кілька способів знайти кращі рішення даної проблеми:

1. У цьому конкретному випадку ми могли б використовувати вбудовану в Python функцію max (), щоб отримати максимальне значення послідовності.

2. Якщо ми знали, що нам потрібно дотримуватися тимчасових значень у списку [це насправді не потрібно], ми могли б спочатку створити копію вхідних списків, а потім продовжити алгоритм.

3. Використовуйте інший алгоритм, який використовує додаткову тимчасову змінну, а не зловживає списком для цього. Наприклад:

```
def mymax ( s ):
    assert len( s ) > 0 , " mymax () arg is an empty sequence "
    tmp = s [0]
    for item in s :
        if item > tmp :
            tmp = item
    return tmp
```

4. Ми могли б передати кортеж (замість списку) функції: кортеж є незмінним і, отже, ніколи не може бути змінений (це призведе до винятку, коли функція намагається записати в елементи в кортежі).

3.4.6 Копіювання об’єктів

Python забезпечує функцію id (), яка повертає ціле число, унікальне для кожного об’єкта. (У поточній реалізації CPython це адреса пам’яті.) Ми можемо використовувати це, щоб визначити, чи однакові два об’єкти. Щоб скопіювати об’єкт послідовності (включаючи списки), ми можемо нарізати його, тобто якщо a є списком, тоді [:] поверне копію a. Ось демонстрація:

```
>>> a = range (10)
a
>>> [0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9]
>>> b = a
>>> b [0] = 42
>>> a                                   # changing b changes a
[42 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9]
>>> id( a )
4327533384
```

```
>>> id( b )
4327533384                              # a and b refer to the same object
>>> c = a [:]
>>> id( c )                              # c is a different object
4327533816
>>> c [0] = 100
>>> a                                    # changing c does not affect a
[42 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9]
```

Стандартна бібліотека Python забезпечує модуль копіювання, який забезпечує функції копіювання, які можна використовувати для створення копій об'єктів. Ми могли використати import copy; c = copy.deepcopy(a) замість c = a [:].

3.5 Рівність та ідентичність / однаковість

Пов'язане питання стосується рівності об'єктів.

3.5.1 Рівність

Оператори ==,> =, <= та! = порівнюють значення двох об'єктів. Об'єкти не повинні мати однаковий тип. Наприклад:

```
>>> a = 1.0; b = 1
>>> type ( a )
<type ‘ float ‘ >
>>> type ( b )
<type ‘ int ‘ >
>>> a == b
True
```

Так оператор == перевіряє, чи рівні значення двох об'єктів.

3.5.2 Ідентичність / однаковість

Щоб перевірити, чи однакові два об'єкти a і b однакові (тобто a і b є посиланнями на одне і те ж місце в пам'яті), ми можемо використовувати оператор is (продовження з прикладу вище):

```
>>> a is b
False
```

Звичайно, вони тут різні, оскільки вони не однотипні. Ми також можемо задати функцію id, яка відповідно до рядка документації в Python 2.7 “Повертає ідентичність об’єкта. Це гарантовано буде унікальним серед одночасно існуючих об’єктів. (Підказка: це адреса пам’яті об’єкта.) ”

```
>>> id( a )
4298197712
>>> id( b )
4298187624
```

який показує, що a і b зберігаються в різних місцях пам’яті.

ТЕМА 4. СПОСТЕРЕЖЕННЯ

Код Python може задавати та відповідати на запитання про себе та об'єкти, якими він маніпулює

4.1 dir()

dir () – це вбудована функція, яка повертає список усіх імен, що належать до деякого простору імен.

1. Якщо жодним аргументам не передається dir (тобто dir ()), він перевіряє простір імен, у якому він був викликаний.

2. Якщо dir отримує аргумент (тобто dir (<object>), тоді він перевіряє простір імен об'єкта, якому він переданий.

Наприклад:

```
>>> apples = [ ' Cox ' , ' Braeburn ' , ' Jazz ']
>>> dir( apples )
[ ' __add__ ' , ' __class__ ' , ' __contains__ ' , ' __delattr__ ' , ' __delitem__ ' ,
' __delslice__ ' , ' __doc__ ' , ' __eq__ ' , ' __ge__ ' , ' __getattribute__ ' ,
' __getitem__ ' , ' __getslice__ ' , ' __gt__ ' , ' __hash__ ' , ' __iadd__ ' ,
' __imul__ ' , ' __init__ ' , ' __iter__ ' , ' __le__ ' , ' __len__ ' , ' __lt__ ' ,
' __mul__ ' , ' __ne__ ' , ' __new__ ' , ' __reduce__ ' , ' __reduce_ex__ ' ,
' __repr__ ' , ' __reversed__ ' , ' __rmul__ ' , ' __setattr__ ' , ' __setitem__ ' ,
' __setslice__ ' , ' __str__ ' , ' append ' , ' count ' , ' extend ' , ' index ' ,
' insert ' , ' pop ' , ' remove ' , ' reverse ' , ' sort ']
>>> dir ()
[ ' __builtins__ ' , ' __doc__ ' , ' __name__ ' , ' apples ']
>>> name = " Peter "
>>> dir( name )
[ ' __add__ ' , ' __class__ ' , ' __contains__ ' , ' __delattr__ ' , ' __doc__ ' ,
' __eq__ ' , ' __ge__ ' , ' __getattribute__ ' , ' __getitem__ ' ,
' __getnewargs__ ' , ' __getslice__ ' , ' __gt__ ' , ' __hash__ ' , ' __init__ ' ,
' __le__ ' , ' __len__ ' , ' __lt__ ' , ' __mod__ ' , ' __mul__ ' , ' __ne__ ' ,
' __new__ ' , ' __reduce__ ' , ' __reduce_ex__ ' , ' __repr__ ' , ' __rmod__ ' ,
' __rmul__ ' , ' __setattr__ ' , ' __str__ ' , ' capitalize ' , ' center ' , ' count ' ,
' decode ' , ' encode ' , ' endswith ' , ' expandtabs ' , ' find ' , ' index ' ,
```

```
' isalnum ' , ' isalpha ' , ' isdigit ' , ' islower ' , ' isspace ' , ' istitle ' ,
' isupper ' , ' join ' , ' ljust ' , ' lower ' , ' lstrip ' , ' replace ' , ' rfind ' ,
' rindex ' , ' rjust ' , ' rsplit ' , ' rstrip ' , ' split ' , ' splitlines ' ,
' startswith ' , ' strip ' , ' swapcase ' , ' title ' , ' translate ' , ' upper ' ,
' zfill ']
```

4.1.1 Магічні імена

Ви знайдете багато імен, які починаються і закінчуються подвійним підкресленням (наприклад, ім'я). Вони називаються магічними іменами. Функції з магічними іменами забезпечують реалізацію певної функціональності python. Наприклад, застосування str до об'єкта a, тобто str (a), will–internally–result у методі a. str () викликається. Цей метод str зазвичай потребує повернення рядка. Ідея полягає в тому, що метод str () повинен бути визначений для всіх об'єктів (включаючи ті, що походять від нових класів), щоб усі об'єкти (незалежно від їх типу або класу) могли бути надруковані за допомогою функції str () . Потім фактичне перетворення деякого об'єкта x у рядок здійснюється за допомогою специфічного для об'єкта методу x.str (). Ми можемо це продемонструвати, створивши клас my_int (int), який наслідується від цілочисельного базового класу Python, і замінює метод str.

```
class my_int (int ):
        """ Inherited from int """
        def __str__ ( self ):
              """ Tailored str representation of my int """
        return " my_int : % s " % (int . __str__ ( self ))

a = my_int (3)
b = int (4)                          # equivalent to b = 4
print " a * b = " , a * b
print " Type a = " , type ( a ) , " str ( a )  =  "  ,  str( a )
print " Type b = " , type ( b ) , " str ( b )  =  "  ,  str( b )
```

Ця програма видає такі результати:

```
a * b = 12
Type a = <class ' __main__ . my_int ' > str ( Sa )= my_int : 3
Type b = <type ' int ' > str ( Sb )= 4
```

4.2 type

Команда type (<object>) повертає тип об’єкта:

```
>>> type (1)
<type ‘ int ‘ >
>>> type (1.0)
<type ‘ float ‘ >
>>> type ( " Python " )
<type ‘ str ‘ >
>>> import math
>>> type ( math )
<type ‘ module ‘ >
>>> type ( math . sin )
<type ‘ builtin_function_or_method ‘ >
```

4.3 isinstance

isinstance(<object>, <typespec>) повертає True якщо даний об’єкт є прикладом даного типу або будь-якого з його суперкласів. Використовуйте е help(isinstance) для повного синтаксису.

```
>>> isinstance (2 , int )
True
>>> isinstance (2. , int )
False
>>> isinstance (a ,int )        # a is an instance of my_int
                                 # which inherits from int
True
>>> type ( a )
<class ‘ __main__ . my_int ‘ >
```

ТЕМА 5. INPUT AND OUTPUT

У цьому розділі ми описуємо старий стиль друку (до Python 3), що включає використання команди print без дужок (що заборонено до використання в Python 3.x) та специфікатори формату старого стилю% (який можна використовувати в Python 2.x та 3.x).

5.1 Друк на стандартному виводі (зазвичай на екрані)

Команда друку – це найбільш часто використовувана команда для друку інформації на «стандартних пристроях виводу», що зазвичай є екраном. Існує два режими використання друку.

5.1.1 Простий друк (не сумісний з Python 3.x)

Найпростіший спосіб використання команди print - це перелік змінних, які слід надрукувати, розділені комами. Ось кілька прикладів:

```
>>> a = 10
>>> b = ‘ test text ‘
>>> print a
10
>>> print b
test text
>>> print a , b
10 test text
>>> print " The answer is " ,a
The answer is 10
>>> print " The answer is " ,a , " and the string contains " ,b
The answer is 10 and the string contains test text
>>> print " The answer is " ,a , " and the string reads " ,b
The answer is 10 and the string reads test text
```

Python додає пробіл між кожним об’єктом, який друкується. Python друкує новий рядок після кожного оператора друку. Щоб заборонити це, додайте зайву кому в кінці рядка:

```
print " Printing in line one " ,
print " ... still printing in line one . "
```

5.1.2 Форматований друк

Більш витончений спосіб форматування виводу використовує синтаксис, дуже схожий на fprintf Matlab (і тому також подібний на printf C). Загальна структура полягає в тому, що існує рядок, що містить специфікатори формату, за яким слідують знак відсотка та кортеж, що містить змінні, які слід надрукувати замість специфікаторів формату.

```
>>> print " a = % d b = % d " % (10 ,20)
a = 10 b = 20
```

Рядок може містити ідентифікатори формату (наприклад, %f для форматування з плаваючою формою, %d для форматування як цілих чисел, і %s для форматування у вигляді рядка):

```
>>> from math import pi
>>> print " Pi = %5.2 f " % pi
Pi = 3.14
>>> print " Pi = %10.3 f " % pi
Pi = 3.142
>>> print " Pi = %10.8 f " % pi
Pi = 3.14159265
>>> print " Pi = % d " % pi
Pi = 3
```

Специфікатор формату типу % W.Df означає, що число з плаваючою комою слід друкувати із загальною шириною з W символів до крапи і D цифр за десятковою крапкою. (Це ідентично Matlab та C, наприклад.) Щоб надрукувати більше одного об’єкта, надайте декілька специфікаторів формату та перелічіть кілька об’єктів у кортежі:

```
>>> print " Pi = %f , 142* pi = % f and pi ^2 = % f . " % ( pi ,142* pi , pi **2)
Pi = 3.141593 , 142* pi = 446.106157 and pi ^2 = 9.869604.
```

Зверніть увагу, що перетворення специфікатора формату та набору змінних у рядок не покладається на команду print:

```
>>> from math import pi
```

```
>>> " pi = % f " % pi
‘ pi = 3.141593 ‘
```

Розглянемо часто використовувані специфікатори формату, що використовують астрономічну одиницю, як приклад:

```
>>> AU = 149597870700 # astronomical unit [ m ]
>>> " % f " % AU # line 1 in table
‘ 149597870700.000000 ‘
```

specifier	style	Example output for AU
%f	floating point	149597870700.000000
%e	exponential notation	1.495979e+11
%g	shorter of %e or %f	1.49598e+11
%d	integer	149597870700
%s	str()	149597870700
%r	repr()	149597870700L

5.1.3 “str” та “str”

Усі об’єкти в Python повинні забезпечувати метод __str__, який повертає приємне рядкове представлення об’єкта. Цей метод __a.str__() викликається, коли ми застосовуємо функцію str до об’єкта a:

```
>>> a = 3.14
>>> a . __str__ ()
‘ 3.14 ‘
>>> str( a )
‘ 3.14 ‘
```

Функція str надзвичайно зручна, оскільки дозволяє друкувати більш складні об’єкти, такі як

```
>>> b = [3 , 4.2 , [ ‘ apple ‘ , ‘ banana ‘] , (0 , 1)]
>>> str( b )
```

```
" [3 , 4.2 , [ ' apple ', ' banana '] , (0 , 1)] "
```

Спосіб друку Python у тому, що він використовує __str__ метод як об’єкт списку. Це надрукує початкову квадратну дужку [і потім викличе метод __str__ першого об’єкта, тобто ціле число 3. Це дасть 3. Потім метод __str__ об’єкта списку друкує кому і переходить до виклику методу __str__ наступного елемента у списку (тобто 4.2) для друку. Рядковий метод об’єкта x викликається неявно, коли ми

• використовуємо специфікатор формату "% s" для друку x

• передаємо об’єкт x безпосередньо команді print:

```
>>> print ( b )
[3 , 4.2 , [ ' apple ' , ' banana '] , (0 , 1)]
>>> print ( " % s " % b )
[3 , 4.2 , [ ' apple ' , ' banana '] , (0 , 1)]
```

5.1.4 “repr” і “ __repr__ ”

Друга функція, repr, повинна перетворити даний об’єкт у рядкову презентацію, щоб цей рядок міг бути використаний для повторного створення об’єкта за допомогою функції eval. Функція repr зазвичай надає більш детальний рядок, ніж str. Застосування repr до об’єкта x спробує здійснити виклик x.__repr__().

```
>>> from math import pi as a1
>>> str( a1 )
' 3.14159265359 '                    # convenient presentation as string
>>> repr ( a1 )
' 3.141592653589793 '                # exact representation as string
>>> number_as_string = repr ( a1 )
>>> a2 = eval ( number_as_string )   # evaluate string
>>> a2
3.141592653589793
>>> a2 - a1                          # -> repr is exact representation
0.0
>>> a1 - eval ( repr ( a1 ))
0.0
>>> a1 - eval (str ( a1 ))           # -> str has lost a few digits
```

```
-2.0694557179012918 e -13
```

Ми можемо перетворити об’єкт на його str() або repr, використовуючи специфікатори формату %s та %r, відповідно.

```
>>> import math
>>> " % s " % math . pi
‘ 3.14159265359 ‘
>>> " % r " % math . pi
‘ 3.141592653589793 ‘
```

5.1.5 Зміни з Python 2 на Python 3: друк

Однією (можливо, найбільш очевидною) зміною, яка переходить з Python 2 на Python 3, є те, що команда print втрачає свій особливий статус. У Python 2 ми могли б надрукувати ”Hello World”, використовуючи

```
print " Hello World "                              # valid in Python 2. x
```

Фактично ми викликаємо функцію print з аргументом “Hello World”. Усі інші функції в Python називаються такими, що аргумент укладається в дужки, тобто

```
print ( " Hello World " )                          # valid in Python 3. x
```

Це нова умова, необхідна в Python 3 (і дозволена для останньої версії Python 2. x.) Хороша новина полягає в тому, що все, що ми дізналися про форматування рядків за допомогою оператора відсотка, також може бути використане в Python 3.x:

```
>>> import math
>>> a = math . pi
>>> " my pi = % f " % a                            # string formatting
‘ my pi = 3.141593 ‘
>>> print " my pi = % f " % a                      # valid print in 2. x
my pi = 3.141593
>>> print ( " my pi = % f " % a )                  # valid print in 2.7 and 3. x
my pi = 3.141593

# another example to follow
>>> " Short pi = %.2 f , longer pi = %.12 f . " % (a , a )
‘ Short pi = 3.14 , longer pi = 3.141592653590. ‘
```

```
>>> print " Short pi = %.2 f , longer pi = %.12 f . " % (a , a )
Short pi = 3.14 , longer pi = 3.141592653590.
>>> print ( " Short pi = %.2 f , longer pi = %.12 f . " % (a , a ))
Short pi = 3.14 , longer pi = 3.141592653590.
```

5.2 Читання та запис файлів

Ось програма, яка

1. пише деякий текст у файл з іменем test.txt,
2. а потім знову читає текст і
3. друкує його на екран.

```
# 1. Write a file
out_file = open ( " test . txt " , " w " )                    # ‘w ‘ stands for Writing
out_file . write ( " Writing text to file . This is the first line .\ n " +\
                    " And the second line . " )
out_file . close ()                                           # close the file

# 2. Read a file
in_file = open ( " test . txt " , " r " )                      # ‘r ‘ stands for Reading
text = in_file . read ()                                      # read complete file
into
                                                              #   string   variable
text
in_file . close ()                                            # close the file

# 3. Display data
print text
```

Дані, що зберігаються у файлі test.txt, є

```
Writing text to file . This is the first line .
And the second line .
```

Більш докладно, ви відкрили файл за допомогою команди open і призначили цей об’єкт відкритого файлу файлу out_file. Потім ми записали дані у файл, використовуючи метод out_file.write. Зверніть увагу, що у наведеному вище прикладі ми надали рядок методу запису. Наприклад, написати цей файл із таблицею імен table.txt

```
1    x    17    =    17
2    x    17    =    34
3    x    17    =    51
4    x    17    =    68
5    x    17    =    85
6    x    17    =   102
7    x    17    =   119
8    x    17    =   136
9    x    17    =   153
10   x    17    =   170
```

ми можемо використовувати цю програму Python

```
f = open ( ‘ table . txt ‘ , ‘w ‘)
for i in range (1 , 11):
    f . write ( " %2 d x 17 = %4 d \ n " % (i , i * 17))
f . close ()
```

Корисно close() файли, коли ми закінчили читати та писати. Якщо завершити роботу Python контрольованим способом (тобто не через відключення електроенергії або помилку в мові Python або операційній системі), вона закриє всі відкриті файли. Тому закрити їх якомога швидше - це кращий шлях.

5.2.1 Приклади читання файлів

Для прикладів нижче ми використовуємо файл myfile.txt, що містить наступні 3 рядки тексту:

This is the first line.

This is the second line.

This is a third and last line

fileobject.read()

Метод fileobject.read () читає весь файл і повертає його у вигляді одного рядка (включаючи нові символи рядка).

```
>>> f = open ( ‘ myfile . txt ‘ , ‘r ‘)
>>> f . read ()
‘ This is the first line .\ nThis is the second line .\ nThis is a third and last line .\ n ‘
```

```
>>> f . close ()
```

fileobject.readlines ()

Метод fileobject.readlines () повертає список рядків, де кожен елемент списку відповідає одному рядку в рядку:

```
>>> f = open ( ‘ myfile . txt ‘ , ‘r ‘)
>>> f . readlines ()
[ ‘ This is the first line .\ n ‘ , ‘ This is the second line .\ n ‘ ,
‘ This is a third and last line .\ n ‘]
>>> f . close ()
```

Це часто використовується для перебору рядків та виконання чогось із кожним рядком. Наприклад:

```
f = open ( ‘ myfile . txt ‘ , ‘r ‘)
for line in f . readlines ():
    print ( " % d characters " % len( line ))
f . close ()
```

з виходом

```
24 characters
25 characters
31 characters
```

Зауважте, що буде зчитувати весь файл у список рядків, коли буде викликаний метод readlines(). Це не проблема, якщо ми знаємо, що файл невеликий і вміститься в пам’яті машини. Якщо так, ми також можемо закрити файл, перш ніж обробляти дані, тобто:

```
f = open ( ‘ myfile . txt ‘ , ‘r ‘)
lines = f . readlines ():
f . close ()
for line in lines :
    print ( " % d characters " % len( line ))
```

Ітерація над рядками (об’єкт файлу)

Існує більш акуратна можливість читати файл, рядок за рядком, який (I) буде читати лише по одному рядку (і, отже, підходить також для

великих файлів) і (II) призводить до отримання більш компактного коду:

```
f = open ( ‘ myfile . txt ‘ , ‘r ‘)
for line in f :
    print ( " % d characters " % len( line ))
f . close ()
```

Тут обробник файлу f діє як в ітераторі і повертатиме наступний рядок у кожній наступній ітерації циклу for, поки не буде досягнуто кінець файлу (а потім цикл for буде припинено).

ТЕМА 6. КОНТРОЛЬ ПОТОКУ

6.1 Основи

Для даного файлу з програмою python інтерпретатор python запуститься вгорі, а потім обробить файл. Ми демонструємо це за допомогою простої програми, наприклад:

```
def f ( x ):
    """ function that computes and returns x * x """
    return x * x

print ( " Main program starts here " )
print ( " 4 * 4 = % s " % f (4))
print ( " In last line of program -- bye " )
```

Основне правило полягає в тому, що команди у файлі (або функції або будь-якій послідовності команд) обробляються зверху вниз. Якщо в одному рядку подано кілька команд (розділених значками ;), то вони обробляються зліва направо (хоча не рекомендується мати кілька операторів у рядку, щоб підтримувати хорошу читабельність коду.) У цьому прикладі інтерпретатор запускає вгорі (рядок 1). Він знаходить ключове слово def і пам'ятає на майбутнє, що тут визначена функція f. (Він ще не виконає тіло функції, тобто рядок 3 – це трапляється лише тоді, коли ми викликаємо функцію.) Інтерпретатор може бачити з відступу, де тіло функції зупиняється: відступ у рядку 5 відрізняється від відступу у першому рядоку у тілі функції (рядок 2), і, отже, тіло функції закінчилося, і виконання повинно тривати з цим рядком. (Порожні рядки не мають значення для цього аналізу.) У рядку 5 інтерпретатор надрукує вихідні дані. Основна програма починається тут. Потім виконується рядок 6. Він містить вираз f (4), який викликатиме функцію f (x), визначену в рядку 1, де x прийме значення 4. [Насправді x є посиланням на об'єкт 4.] Потім виконується функція f обчислює і повертає 4 * 4 у рядку 3. Це значення 16 використовується у рядку 6 для заміни f (4), а

потім рядкове представлення %s об’єкта 16 друкується як частина команди друку у рядку 6. Інтерпретатор потім переходить до рядка 7 до закінчення програми. Зараз ми дізнаємося про різні можливості подальшого управління цього потоку.

6.1.1 Умови

Значення python True і False - це спеціальні вбудовані об’єкти:

```
>>> a = True
>>> print a
True
>>> type ( a )
<type ‘ bool ‘ >
>>> b = False
>>> print b
False
>>> type ( b )
<type ‘ bool ‘ >
```

Ми можемо оперувати цими двома логічними значеннями, використовуючи булівську логіку, наприклад логіку та операцію (і):

```
>>> True and True                                   # logical and operation
True
>>> True and False
False
>>> False and True
False
>>> True and True
True
>>> c = a and b
>>> print c
False
```

Є також логічне або (or) і заперечення (not):

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

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

У комп'ютерному коді нам часто потрібно обчислити якийсь вираз, який є або true, або false (іноді його називають "Присудок"). Наприклад:

```
>>> x = 30                              # assign 30 to x
>>> x > 15                              # is x greater than 15
True
>>> x > 42
False
>>> x == 30                             # is x the same as 30?
True
>>> x == 42
False
>>> not x == 42                         # is x not the same as 42?
True
>>> x != 42                             # is x not the same as 42?
True
>>> x > 30                              # is x greater than 30?
False
>>> x >= 30                             # is x greater than or equal to 30?
True
```

6.2 If-then-else

Оператор if дозволяє умовне виконання коду, наприклад:

```
a = 34
if a > 0:
    print " a is positive "
```

if також може мати гілку else який виконується, якщо умова помилкова:

```
a = 34
if a > 0:
    print " a is positive "
else :
```

```
    print " a is non - positive ( i . e . negative or zero ) "
```

Нарешті, є elif (читати як ключове слово “else if”), що дозволяє перевірити кілька (ексклюзивних) можливостей:

```
a = 17
if a == 0:
    print " a is zero "
elif a < 0:
    print " a is negative "
else :
    print " a is positive "
```

6.3 Цикл for

Цикл for дозволяє перебирати послідовність (наприклад, це може бути рядок або список). Ось приклад:

```
>>> for animal in [ ‘ dog ‘ , ‘ cat ‘ , ‘ mouse ‘ ]:
...        print animal , animal . upper ()
dog DOG
cat CAT
mouse MOUSE
```

Разом з командою range () можна перебирати зростаючі цілі числа:

```
>>> for i in range (5 ,10):
...          print i
5
6
7
8
9
```

6.4 Цикл while

Ключове слово while дозволяє повторити операцію, якщо умова є істинні. Припустимо, ми хотіли б знати, скільки років нам потрібно зберігати 100 фунтів стерлінгів на ощадному рахунку, щоб досягти 200 фунтів просто

завдяки щорічній виплаті відсотків за ставкою 5%. Ось програма для підрахунку, що це займе 15 років:

```
mymoney = 100                          # in GBP
rate = 1.05                            # 5% interest
years = 0
while mymoney < 200:                   # repeat until 20 pounds reached
        mymoney = mymoney * rate
        years = years + 1
print ‘ We need ‘ , years , ‘ years to reach ‘ , mymoney , ‘ pounds . ‘
```

обчислить

```
We need 15 years to reach 207.892817941 pounds .
```

6.5 Реляційні оператори (порівняння) у операторах if і while

Загальна форма операторів if та циклів while однакова: після ключового слова if або while існує умова, після якої ставиться двокрапка. У наступному рядку починається новий (і, отже, з відступом!) блок команд, який виконується, якщо умова має значення True). Наприклад, умовою може бути рівність двох змінних a1 і a2, що виражається як a1 == a2:

```
a1 = 42
a2 = 42
if a1 == a2 :
   print ( " a1 and a2 are the same " )
```

Іншим прикладом є перевірити, чи не однакові a1 та a2. Для цього ми маємо дві можливості. Варіант номер 1 використовує оператор нерівності! =:

```
if a1 != a2 :
   print ( " a1 and a2 are different " )
```

Варіант другий використовує ключове слово, яке не стоїть перед умовою:

```
if not a1 == a2 :
    print ( " a1 and a2 are different " )
```

Порівняння «більших» (>), «менших» (<), «більших рівних» (>=) та «менших рівних» (<=) є простими. Нарешті, ми можемо використовувати логічні оператори «і» і «або» для комбінування умов:

```
if a > 10 and b > 20:
    print " A is greater than 10 and b is greater than 20 "
if a > 10 or b < -5:
    print " Either a is greater than 10 , or " \
" b is smaller than -5 , or both . "
```

Використовуйте підказку Python, щоб експериментувати з цими порівняннями та логічними виразами. Наприклад:

```
>>> T = -12.5
>>> if T < -20:
...         print " very cold "
...
>>> if T < -10:
...         print " quite cold "
...
quite cold
>>> T < -20
False
>>> T < -10
True
>>>
```

6.6 Винятки

Навіть якщо вираз або вираз синтаксично правильні, це може спричинити помилку при спробі його виконати. Помилки, виявлені під час виконання, називаються винятками і не обов’язково є фатальними: винятки можна ловити та вирішувати в рамках програми. Однак більшість винятків не обробляються програмами, і вони призводять до повідомлень про помилки, як показано тут

```
>>> 10 * (1/0)
Traceback ( most recent call last ):
  File " < stdin > " , line 1 , in ?
ZeroDivisionError : integer division or modulo by zero
>>> 4 + spam *3
Traceback ( most recent call last ):
```

```
    File " < stdin > " , line 1 , in ?
NameError : name ‘ spam ‘ is not defined
>>> ‘2 ‘ + 2
Traceback ( most recent call last ):
    File " < stdin > " , line 1 , in ?
TypeError : cannot concatenate ‘ str ‘ and ‘ int ‘ objects
```

Схематичне вилучення винятків з усіма опціями спробуйте:

```
try:
    # code body
except ArithmeticError :
    # what to do if arithmetic error
except IndexError , the_exception :
    # the_exception refers to the exeption in this block
except :
    # what to do for ANY other exception
else : # optional
    # what to do if no exception raised
try:
    # code body
finally :
    # what to do ALWAYS
```

Починаючи з Python 2.5, ви можете використовувати оператор with для спрощення написання коду для деяких заздалегідь визначених функцій, зокрема функції open для відкриття файлів. Приклад: Ми намагаємося відкрити файл, який не існує, і Python викличе виняток типу IOError, що означає Вхідна вихідна помилка:

```
>>> f = open ( " filenamethatdoesnotexist " , " r " )
Traceback ( most recent call last ):
    File " < stdin > " , line 1 , in < module >
IOError : [ Errno 2] No such file or directory : ‘ filenamethatdoesnotexist ‘
```

Якщо ми писали програму з користувацьким інтерфейсом, де користувач повинен ввести або вибрати ім’я файлу, ми не хотіли б, щоб програма зупинялася, якщо файл не існує. Натомість нам потрібно вловити цей виняток і діяти відповідно (наприклад, повідомивши користувача, що

файл з таким ім’ям файлу не існує, і запитати, чи хочуть вони спробувати інше ім’я файлу). Ось скелет для лову цього винятку:

```
>>> try:
...         f = open ( " filenamethatdoesnotexist " ," r " )
...     except IOError :
...         print " Could not open that file "
...
Could not open that file
```

6.6.1 Створення винятків

Можливості створення винятку

– raise OverflowError

– raise OverflowError, "Bath is full" (Old style, now discouraged)

– raise OverflowError("Bath is full")

– e = OverflowError("Bath is full"); raise e

Ієрархія винятків

Стандартні винятки організовані в ієрархії успадкування, наприклад OverflowError – це підклас ArithmeticError (не BathroomError); це можна побачити, наприклад, викликавши help(‘exceptions’). Ви можете отримати власні винятки з будь-яких стандартних. Це гарний стиль, коли кожен модуль визначає власний базовий виняток

6.6.2 Створення власних винятків

Ви можете і повинні отримувати власні винятки із вбудованого винятку.

Щоб побачити, які вбудовані винятки існують, загляньте у винятки модулів (спробуйте help(‘exceptions’))

6.6.3 LBYL проти EAFP

• LBYL (Сім разів відміряй, один раз відріж) проти

• EAFP (Простіше просити прощення, ніж дозвіл)

Приклад для LBYL:

```
if denominator == 0:
    print " Oops "
else :
    print numerator / denominator
```

Простіше просити прощення, ніж дозволити:

```
try:
    print numerator / denominator
except ZeroDivisionError :
    print " Oops "
```

У документації Python сказано про EAFP: простіше просити прощення, ніж дозвіл. Цей поширений стиль кодування Python передбачає існування дійсних ключів або атрибутів і вловлює винятки, якщо припущення виявляється хибним. Цей чистий і швидкий стиль характеризується наявністю багатьох спроб, крім висловлювань. Ця техніка контрастує зі стилем LBYL, загальним для багатьох інших мов, таких як C.

У документації Python сказано про LBYL: Сім разів відміряй, один раз відріж. Цей стиль кодування явно перевіряє попередні умови перед здійсненням дзвінків або пошуку. Цей стиль контрастує з підходом EAFP і характеризується наявністю багатьох тверджень if.

ТЕМА 7. ФУНКЦІЇ ТА МОДУЛІ

7.1 Вступ

Функції дозволяють групувати ряд операторів у логічний блок. Ми взаємодіємо з функцією через чітко визначений інтерфейс, забезпечуючи певні параметри функції та отримуючи деяку інформацію назад. Окрім цього інтерфейсу, ми зазвичай не знаємо, як саме функція виконує роботу, щоб отримати значення, яке вона повертає. Наприклад, функція math.sqrt: ми не знаємо, як саме вона обчислює квадратний корінь, але ми знаємо про інтерфейс: якщо ми передаємо x у функцію, вона поверне (наближення) $\sqrt{x}$. Ця абстракція є корисною річчю: це загальноприйнята техніка розбиття системи на менші (чорні ящики) компоненти, які всі працюють разом через чітко визначені інтерфейси, але яким не потрібно знати про внутрішню реалізацію функціональності. Насправді, відсутність необхідності дбати про ці деталі реалізації може допомогти отримати чіткіший погляд на систему, що складається з багатьох з цих компонентів. Функції забезпечують основні блоки функціональних можливостей у більших програмах (та комп'ютерному моделюванні) та допомагають контролювати властиву складність процесу. Ми можемо згрупувати функції разом у модуль Python і таким чином створити власні бібліотеки функціональних можливостей.

7.2 Використання функцій

Слово «функція» має різне значення в математиці та програмуванні. У програмуванні воно відноситься до іменованої послідовності операцій, які виконують обчислення. Наприклад, функція sqrt (), яка визначена в математичному модулі, обчислює квадратний корінь заданого значення:

```
>>> from math import sqrt
>>> sqrt (4)
2.0
```

Значення, яке ми передаємо функції sqrt, дорівнює 4 в цей приклад. Це значення називається аргументом функції. Функція може мати більше одного аргументу. Функція повертає значення 2.0 (результат його обчислення) до “контексту виклику”. Це значення називається поверненим значенням функції. Зазвичай прийнято говорити, що функція приймає аргумент і повертає результат або повернене значення.

Поширена плутанина щодо друку та повернення значень

Поширеною помилкою для початківців є плутання друку значень із повертаючими значеннями. У наступному прикладі неможливо зрозуміти, чи повертає функція math.sin значення, чи друкує значення:

```
>>> import math
>>> math . sin (2)
0.90929742682568171
>>>
```

Ми імпортуємо математичний модуль і викликаємо функцію math.sin з аргументом 2. Виклик math.sin (2) фактично поверне значення 0.909 ..., не надрукує його. Однак, оскільки ми не призначили значення змінної змінної, підказка Python надрукує повернутий об’єкт. Наступна альтернативна послідовність працює, лише якщо повертається значення:

```
>>> x = math . sin (2)
>>> print x
0.909297426826
```

Зворотне значення виклику функції math.sin (2) присвоюється змінній x, а x друкується в наступному рядку. Як правило, функції повинні виконуватися "тихо" (тобто нічого не друкувати) і повідомляти результат їх обчислення через повернене значення. Частина плутанини щодо надрукованих та повернутих значень у підказці Python походить від друку підказки Python (подання) повернутих об’єктів, якщо повернуті об’єкти не визначено. Як правило, бачити повернені об’єкти - це саме те, що ми хочемо (оскільки ми зазвичай дбаємо про повернутий об’єкт), просто під час

вивчення Python це може спричинити легку плутанину щодо функцій, що повертають значення або друкують значення.

7.3 Визначення функцій

Загальний формат визначень функції:

```
def my_function ( arg1 , arg2 , ... , argn ):
        """ Optional docstring . """

        # Implementation of the function

        return result     # optional
# this is not p
some_command
```

Деяка термінологія розрізняє функції, які повертають значення, і функції, які не повертають значення. Різниця стосується того, чи функція надає повернене значення (= плідне), чи функція явно не повертає значення (= безрезультатне). Якщо функція не використовує оператор return, ми схильні говорити, що функція нічого не повертає (тоді як насправді in завжди повертає об'єкт None, коли він закінчується - навіть якщо оператор return відсутній). Наприклад, функція greeting надрукує "Hello World" при виклику (і безрезультатно, оскільки не повертає значення).

```
def greeting ():
    print " Hello World ! "
```

Якщо ми викликаємо цю функцію:

```
>>> greeting ()
Hello World !
```

він друкує "Hello World" у формат stdout, як ми і очікували. Якщо ми присвоїмо значення функції, що повертається, змінній x, ми можемо перевірити її згодом:

```
>>> x = greeting ()
Hello World !
>>> print x
```

```
None
```

та виявить, що функція привітання справді повернула об'єкт None. Іншим прикладом для функції, яка не повертає жодного значення (це означає, що у функції немає ключового слова return), буде:

```
def printpluses ( n ):
    print n * " + "
```

Як правило, функції, які повертають значення, є більш корисними, оскільки вони можуть використовуватися для складання коду (можливо, як інша функція), розумно їх поєднуючи. Давайте розглянемо кілька прикладів функцій, які повертають значення. Припустимо, нам потрібно визначити функцію, яка обчислює квадрат заданої змінної. Джерелом функції може бути:

```
def square ( x ):
    return x * x
```

Ключове слово def повідомляє Python, що ми визначаємо функцію в цей момент. Функція приймає один аргумент (x). Функція повертає x*x, що, звичайно, x^2. Ось список файлів, який показує, як функцію можна визначити та використовувати: (зверніть увагу, що цифри ліворуч є номерами рядків і не є частиною програми)

```
1  def square ( x ):
2      return x * x
3
4  for i in range (5):
5      i_squared = square ( i )
6      print i , '* ' , i , '= ' , i_squared
```

Варто зазначити, що рядки 1 і 2 визначають функцію квадрата, тоді як рядки 4 - 6 є основна програма. Ця програма видасть наступний результат

```
0  *  0  =  0
1  *  1  =  1
2  *  2  =  4
3  *  3  =  9
4  *  4  =  16
```

Ми можемо визначити функції, які приймають більше одного аргументу:

```
import math

def hypot (x , y ):
    return math . sqrt ( x * x + y * y )
```

Також можна повернути більше одного аргументу. Ось приклад функції, яка перетворює заданий рядок у всі символи з великої літери та всі символи в нижньому регістрі та повертає дві версії. Ми включили основну програму, щоб показати, як цю функцію можна викликати:

```
def upperAndLower ( string ):
            return string . upper () , string . lower ()

testword = ‘ Banana ‘

uppercase , lowercase = upperAndLower ( testword )

print testword , ‘ in lowercase : ‘ , lowercase ,\
           ‘ and in uppercase ‘ , uppercase
```

Ми можемо визначити кілька функцій Python в одному файлі. Ось приклад з двома функціями:

```
def returnstars ( n ):
     return n * ‘* ‘

def print_centred_in_stars ( string ):
    linelength = 46
    starstring = returnstars (( linelength - len( string )) / 2)

     print starstring + string + starstring

print_centred_in_stars ( ‘ Hello world ! ‘)
```

Ця програма видає

```
**************** Hello world !****************
```

7.4 Значення за замовчуванням та необов'язкові параметри

Python дозволяє визначати значення за замовчуванням для параметрів функції. Ось приклад:

```
def print_mult_table (n , upto =10):
    for i in range (1 , upto + 1):
        print " %3 d * % d = %4 d " % (i , n , i * n )
print_mult_table (5)
print_mult_table (9 , 3)
```

Ця програма надрукує наступні результати при виконанні:

```
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
6 * 5 = 30
7 * 5 = 35
8 * 5 = 40
9 * 5 = 45
10 * 5 = 50
1 * 9 = 9
2 * 9 = 18
3 * 9 = 27
```

То як це працює? Функція print_mult_table приймає два аргументи: n та upto. Перший аргумент n є "нормальною" змінною. Другий аргумент upto має значення за замовчуванням 10. Іншими словами: якщо користувач цієї функції надає лише один аргумент, то це забезпечує значення n і upto буде за замовчуванням 10. Якщо вказано два аргументи, перший буде бути для n, а другий для upto (як показано в прикладі коду вище).

7.5 Модулі

Модулі:

1. Групують функціонал.

2. Забезпечують простори імен.

3. Стандартна бібліотека Python містить велику колекцію модулів – «Batteries Included».

Спробуйте help(‘modules’)

4. Є засобами розширення Python

7.5.1 Імпорт модулів

```
import math
```

Це введе математику імен у простір імен, в якому була видана команда імпорту. Імена в математичному модулі не відображатимуться у просторі імен, що вмикається: до них потрібно отримати доступ за допомогою імені math. Наприклад: math.sin

```
import math , cmath
```

Можна імпортувати в одному операторі більше одного модуля, хоча Python Style Guide рекомендує цього не робити. Натомість ми повинні написати

```
import math
import cmath
```

```
import math as mathematics
```

Ім’я, під яким модуль відомий локально, може відрізнятися від його «офіційного» імені. Типовим використанням цього є:

• Щоб уникнути зіткнень імен із існуючими іменами

• Змінити назву на щось більш кероване. Наприклад, імпортуйте SimpleHTTPServer як shs. Це не рекомендується для виробничого коду (оскільки довші значущі назви роблять програми набагато зрозумілішими, ніж короткі загадкові), але для інтерактивного тестування ідей можливість використання короткого синоніма може значно полегшити ваше життя. Враховуючи, що (імпортовані) модулі є об’єктами першого класу, ви, звичайно, можете просто зробити shs = SimpleHTTPServer, щоб отримати дескриптор, який можна легше набирати на модулі

```
from math import sin
```

Це імпортує функцію sin з математичного модуля, але це не введе ім’я math у поточний простір імен. Це лише введе ім’я sin у поточний простір імен. Можливо, за один раз витягти з модуля більше одного імені:

```
from math import sin , cos
```

Нарешті, давайте подивимось на це позначення:

```
from math import *
```

Ще раз це не вводить ім’я math у поточний простір імен. Однак він вводить усі публічні імена математичного модуля до поточного простору імен. Взагалі кажучи, робити це погано:

1. Багато нових імен буде скинуто в поточний простір імен.
2. Ви впевнені, що вони не зроблять жодних імен, які вже є?
3. Буде дуже важко простежити, звідки взялися ці імена.
4. Деякі модулі (включаючи модулі у стандартній бібліотеці) рекомендують імпортувати їх таким чином. Використовуйте з обережністю!
5. Це чудово для інтерактивного швидкого та брудного тестування або невеликих розрахунків.

7.5.2 Створення модулів

Модуль – це не що інше, як файл python. Ось приклад файлу модуля, який зберігається в module1.py:

```
def someusefulfunction ():
    pass
print " My name is " , __name__
```

Ми можемо виконати цей файл (модуля) як звичайну програму на python (наприклад, python module1.py), і результат

```
My name is __main__
```

Зауважимо, що магічна змінна __name__ приймає значення __main__, якщо файл програми module1.py виконується. З іншого боку, ми можемо імпортувати module1.py в інший файл (який міг би мати назву prog.py), наприклад так:

```
import module1                                   # in file prog . py
```

Коли Python натрапляє на оператор імпорту module1 у програмі prog.py, він шукає файл module1.py у поточному робочому каталозі (і якщо він не може знайти його там у всіх каталогах у sys.path), то відкриває файл module1.py. Під час синтаксичного аналізу файлу module1.py зверху вниз, він додасть будь-які визначення функції в цьому файлі у простір імен module1 у контексті виклику (це основна програма в prog.py). У цьому прикладі є лише функція деякої корисної функції. Після завершення процесу імпорту ми зможемо використовувати module1.someusefulfunction у prog.py. Якщо при імпортуванні module1.py Python зустрічає оператори, крім визначень функції (і класу), він виконує їх негайно. У цьому випадку він, таким чином, зіткнеться із твердженням print "My name is", __name__. Результат цього читається:

```
My name is module1
```

Зверніть увагу на різницю у виведенні, якщо ми імпортуємо module1.py, а не виконуємо його самостійно: __name__ усередині модуля приймає значення імені модуля, якщо файл імпортується.

7.5.3 Використання __name__

Таким чином

– __name__ є " main ", якщо файл модуля запускається самостійно

– __name__ – це ім’я модуля (тобто ім’я файлу модуля .py без суфікса), якщо файл модуля імпортовано. Для цього ми можемо використовувати наступний оператор if у module1.py для написання коду, який запускається лише тоді, коли модуль виконується самостійно:

```
def someusefulfunction ():
        pass
if __name__ == " __main__ " :
        print ( " I am running on my own " )
```

Це корисно для збереження тестових програм або демонстрації здібностей модуля в цій «умовній» основній програмі. Поширеною практикою є наявність у будь-яких файлах модулів такої умовної основної програми, яка демонструє її можливості.

7.5.4 Приклад 1

Наступний приклад показує основну програму для іншого файлу vectools.py, який використовується для демонстрації можливостей функцій, визначених у цьому файлі:

```
from __future__ import division
import math

import numpy as N

def norm ( x ):
        """ returns the magnitude of a vector x """
        return math . sqrt (sum ( x ** 2))

def unitvector ( x ):
        """ returns a unit vector x /| x |. x needs to be a numpy array . """
        xnorm = norm ( x )
        if xnorm == 0:
          raise ValueError ( " Can ‘t normalise vector with length 0 " )
        return x / norm ( x )

if __name__ == " __main__ " :
        # a little demo of how the functions in this module can be used :
        x1 = N . array ([0 , 1 , 2])
        print ( " The norm of " + str( x1 ) + " is " + str( norm ( x1 )) + " . " )
        print ( " The unitvector in direction of " + str( x1 ) + " is " \
              + str( unitvector ( x1 )) + " . " )
```

Якщо цей файл виконується за допомогою python vectools.py, __name__ =="__main__" відповідає істині, а на виході читається

```
The norm of [0 1 2] is 2.2360679775 .
The unitvector in direction of [0 1 2] is [ 0. 0.4472136 0.89442719] .
```

Якщо цей файл імпортується (тобто використовується як модуль) в інший файл python, тоді __name__ =="__main__" хибне, і цей блок операторів не буде виконаний (і вихід не буде створений). Це досить поширений спосіб умовного виконання коду у файлах, що забезпечують бібліотечні функції. Код, який виконується, якщо файл запускається самостійно, часто

складається із серії тестів (щоб перевірити, чи виконують функції файлу правильні операції - регресійні тести або модульні тести), або деяких прикладів того, як функціонує

7.5.5 Приклад 2

Навіть якщо програма Python не призначена для використання як файл модуля, корисно завжди використовувати умовну основну програму:

– часто пізніше виявляється, що функції у файлі можуть бути використані повторно (і тоді зберігає роботу);
– це зручно для регресійного тестування.

Припустимо, вправа дається для написання функції, яка повертає перші 5 простих чисел, і на додаток до їх друку. (Звичайно, для цього існує тривіальне рішення, оскільки ми знаємо прості числа, і нам слід уявити, що необхідне обчислення є більш складним). У когось може виникнути спокуса написати

```
def primes5 ():
    return (2 , 3 , 5 , 7 , 11)

for p in primes5 ():
    print " % d " % p ,
```

Краще використовувати стиль умовної основної функції, тобто:

```
def primes5 ():
    return (2 , 3 , 5 , 7 , 11)

if __name__ == " __main__ " :
    for p in primes5 ():
        print " % d " % p ,
```

може стверджувати, що наступне ще чистіше:

```
def primes5 ():
    return (2 , 3 , 5 , 7 , 11)

def main ():
    for p in primes5 ():
        print " % d " % p ,
```

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

але будь-який із останніх двох варіантів хороший.

Приклад у розділі 9.1 демонструє цю техніку. Включення функцій з іменами, що починаються з тесту, сумісне з дуже корисною системою регресійного тестування py.test.

ПЕРЕЛІК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Костюченко А.О. Основи програмування мовою Python: навч. посіб. Чернігів: ФОП Баликіна С. М., 2020. 180 с.

2. Каштан В.Ю. Програмування комп’ютерних систем мовою Python. Частина 1: навч. наоч. посіб. / В.Ю. Каштан, В.В. Гнатушенко, Д.В. Сущевський, Є.О. Обиденний; М-во освіти і науки України, Нац. техн. ун-т «Дніпровська політехніка». Дніпро: НТУ «ДП», 2024. 189 с.

3. Пол Беррі. Head First, Python. Фабула, Ганна Якубовська, 2-ге вид., 2021. 624с.

4. Luciano Ramalho. Fluent Python, 2nd ed. Published by O’Reilly Media, Inc., April 2022. 1011p.

5. Reuven M. Lerner, Python Workout: 50 ten-minute exercises, 1st ed. Manning Publications., Juli 2020. 248p.

6. Michael Inden. Python Challenges: 100 Proven Programming Tasks Designed to Prepare You for Anything, 1st ed. Apress, April 2022. 691p.

7. Ramalho L. Fluent Python: Clear, Concise, and Effective Programming. 2nd Edition. O’Reilly Media, 2022. 1016 p

8. Eric Matthes. Python Crash Course : A Hands-On, Project-Based Introduction to Programming, 3nd ed. No Starch Press, May 2023. 552p.

9. Patrick Viafore. Robust Python: Write Clean and Maintainable Code 1st ed. Published by O’Reilly Media, Inc., August 2021. 378p.

10. Alice Zhao. SQL Pocket Guide: A Guide to SQL Usage 4th ed. Published by O’Reilly Media, Inc., October 2021. 354p.

П 78 Програмування на Python. Методичні вказівки до практичних занять для здобувачів першого (бакалаврського) рівня вищої освіти галузі знань 12 (F) Інформаційні технології денної та заочної форм навчання / уклад. С.М. Костючко, Л.М. Конкевич. Луцьк: ЛНТУ, 2026. 81 с.

Конспект лекцій призначений для здобувачів першого (бакалаврського) рівня вищої освіти галузі знань 12 (F) Інформаційні технології.

Комп'ютерний набір С.М. Костючко, Л.М. Конкевич

Редактор С.М. Костючко, Л.М. Конкевич

Підп. до друку «___» ________ 2026р.
Формат 60х84/16. Папір офс. Гарнітура Таймс.
Ум. друк. арк. ____. Тираж 10 прим. Зам. ____

Відділ іміджу та промоцій
Луцького національного технічного університету
43018, м. Луцьк, вул. Львівська, 75
ВІП ЛНТУ