ЗАТВЕРДЖЕНО

Науково-методичною радою
Державного університету
«Житомирська політехніка»

протокол від 9 листопада 2020 р.
№4

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ
для проведення практичних робіт з навчальної дисципліни «Обчислювальна техніка та програмування»

для здобувачів вищої освіти освітнього ступеня «бакалавр»
спеціальностей 172 «Телекомунікації та радіотехніка»
та 163 Біомедична інженерія освітньо-професійна програма
«Інформаційні відеосистеми та системи контролю доступу»
освітньо-професійна програма «Телекомунікації та радіотехніка»
освітньо-професійна програма «Біомедичний комп'ютинг»
факультет інформаційно комп'ютерних технологій кафедра
біомедичної інженерії та телекомунікацій

Рекомендовано на засіданні біомедичної інженерії та телекомунікації
31серпня 2020 р., протокол № 9

Розробник: старший викладач кафедри БІ та Т МОРОЗОВ Дмитро

Житомир
2020 – 2021 н.р.

ЗМІСТ

Теоретична частина

Python — це потужна мова програмування, якою легко оволодіти. Вона має ефективні структури даних високого рівня та простий, але ефективний підхід до об'єктно-орієнтованого програмування. Елегантний синтаксис Пайтона, динамічна обробка типів, а також те, що це інтерпретована мова, роблять його ідеальним для написання скриптів та швидкої розробки прикладних програм у багатьох галузях на більшості платформ.

Інтерпретатор мови Пайтон і багата стандартна бібліотека (як код-джерело, так і бінарні дистрибутиви для усіх головних операційних систем) можуть бути отримані з сайту Пайтона, і можуть вільно розповсюджуватися. Цей самий сайт має дистрибутиви та посилання на численні модулі, програми, утиліти та додаткову документацію.

Інтерпретатор мови Пайтон може бути легко розширений функціями та типами даних, розробленими на C чи C++ (або на іншій мові, яку можна викликати із C). Пайтон також зручний як мова сценаріїв що вбудовуються в прикладні програми, для додаткових налаштувань функціональності.

Цей підручник повинен у загальних рисах ознайомити читача з головними концепціями та рисами Пайтона. Працюючи з цим посібником, загалом добре мати інтерпретатор мови Пайтон під рукою, але всі приклади самодостатні, отже цей текст може просто бути прочитаний.

Щодо опису стандартних об'єктів та модулів, див. Python Library Reference. Python Reference Manual дає більш формальне визначення мови. Щоб писати розширення на C та C++, читайте Extending and Embedding the Python Interpreter та Python/C API Reference. Існує також кілька книжок, що детально розглядають Пайтон.

Цей огляд не є всеохопним, у ньому не розглянуто кожну окрему рису чи навіть усі найбільш вживані особливості. Натомість, він містить риси мови, які потребують першочергової уваги, та подає читачеві загальне уявлення про смак та стиль мови. Прочитавши його, ви зможете читати і створювати власні модулі та програми, а також будете готові ознайомитися з різноманітними модулями бібліотеки Пайтона, описаними у Python Library Reference.

Якщо ви колись писали великий скрипт командного рядка, то вам напевно знайоме це почуття: вам хочеться додати ще одну функцію, але скрипт уже й так занадто повільний та заскладний, чи ця функція потребує виклику системної команди або іншої функції, яку можна викликати лише із C... Часто проблема не така вже й складна, щоб переписувати увесь скрипт на C; можливо проблема вимагає рядків змінної довжини чи інших типів даних (таких як впорядковані списки назв файлів), що легко створити в Shell, але які потребують багато роботи на C, чи можливо ви не достатньо обізнані з C.

Інша ситуація: можливо вам потрібно працювати з кількома бібліотеками, написаними на C, але традиційний цикл "написання-компіляція-тестування-компіляція" занадто повільний. Вам потрібно розробити програму швидше. Можливо ви написали програму, яка потребує мови розширення, але вам не хочеться створювати нову мову, писати та налагоджувати її інтерпретатор, і потім прив'язувати його до вашої програми.

У таких випадках Пайтон може бути саме тією мовою, що потрібна. Його легко використовувати, у той же час це справжня мова програмування, що має багатшу структуру та підтримує написання більших програм, ніж традиційна оболонка. З іншого боку, Пайтон пропонує зручніше та детальніше виявлення помилок ніж C, і, будучи мовою дуже високого рівня, він має вмонтовані високорівневі типи даних, зокрема гнучкі масиви та словники, для ефективного створення яких на мові C потрібно кілька днів. Завдяки загальнішим типам даних, Пайтон може бути застосований для вирішення набагато ширшого кола проблем, ніж така мова як AWK, чи навіть Perl; водночас багато речей можуть бути створені на Пайтоні так само просто, як і на цих мовах.

Пайтон дозволяє розбити вашу програму на модулі, які можуть згодом використовуватися в інших програмах, написаних на Пайтоні. Пайтон має велику кількість стандартних модулів, які ви можете покласти в основу своїх програм або на яких можете почати вчитися програмувати. Існують також вбудовані модулі для таких задач як файловий ввід-вивід, системні виклики, сокети, і навіть графічні інтерфейси, наприклад інтерфейс до набору віджетів мови Tk.

Пайтон — це інтерпретована мова, що може зберегти вам чимало часу при розробці програм, тому що компіляція та прив'язування (linking) не потрібні. Інтерпретатор може також використовуватися у діалоговому режимі, що спрощує експериментування з різними рисами мови, написання одноразових програм чи тестування шляхом зворотньої розробки. Це також зручний настільний калькулятор.

Пайтон дозволяє створювати дуже компактні та читабельні програми. Типова програма, написана на Пайтоні — набагато коротша ніж еквівалентна програма на C чи C++, що відбувається з таких причин:

- структури даних високого рівня дозволяють виразити складні операції за допомогою однієї інструкції
- групування інструкцій в блоки робиться за допомогою виділення відступами замість фігурних дужок
- декларація змінних чи аргументів не потрібна

Пайтон можна розширити: якщо ви вмієте програмувати на C, то вам буде достить легко додати нову вбудовану функцію до інтерпретатора, що надасть можливість виконувати критичні операції з максимальною швидкістю, або прив'язати Пайтон до вже скомпільованих бібліотек (наприклад бібліотек графічного інтерфейсу). Також ви можете прив'язяти інтерпретатор Пайтона до програми, написаної на C і використовувати його як розширення чи командну мову для вашої програми.

Між іншим, мова називається на честь комедійного шоу від Бі-Бі-Сі "Monty Python's Flying Circus" ("Летючий цирк Монті Пайтона") і не має жодного відношення до рептилій. Жарти про Монті Пайтона не лише дозволяються, але й заохочуються!

Тепер, коли ви всі вже мабуть полюбили Пайтон, ви напевно хочете докладніше ознайомитися з цією мовою. Оскільки найкращий спосіб вивчити мову — це працювати з нею, отож і ми рекомендуємо вам це зробити якнайшвидше.

У наступному розділі йтиметься про механіку користування інтерпретатором. Це доволі банальна, але істотна для розуміння подальших прикладів інформація.

Решта цього навчального посібника пояснюватиме різні особливості мови й системи Пайтон на прикладах, починаючи з простих виразів, інструкцій і типів даних, через функції і модулі і нарешті до більш складних концепцій, як винятки та визначені користувачем класи.

Виклик інтерпретатора

Якщо інтерпретатор мови Пайтон встановлено, його місцезнаходженням зазвичай є `/usr/local/bin/python`. Якщо додати `/usr/local/bin` до змінної оболонки Юнікса `PATH` (шляхи до файлів що виконуються), то це дасть змогу викликати інтерпретатор за допомогою команди

```
python
```

Місце, де живе інтерпретатор, може бути задане під час інсталяції. Якщо ви не певні, де знаходиться виконуваний файл Пайтон, то виконайте команду: `which python` або запитайте місцевого ґуру чи системного адміністратора (`/usr/local/python`, наприклад, є популярною альтернативою).

Ввід символу "кінець файла" (`Control-D` на Unix, `Control-Z` на Windows) змусить інтерпретатор вийти зі статусом нуль. Якщо це не працює, залишити

інтерпретатор можна за допомогою таких команд:

```
import sys; sys.exit()
```

А часто просто

```
exit()
```

Редагування рядка в інтерпретаторі загалом не дуже складне. На Юніксі той, хто встановив інтерпретатор, міг також додати підтримку бібліотеки GNU для зчитування рядка (GNU readline library), котра надає ширші можливості інтерактивного редагування та пам'ятає історію команд. Чи не найшвидший спосіб перевірити, чи можливе редагування рядка — це введення Control-P відразу після запуску інтерпретатора. Якщо ви чуєте гудок, то редагування рядка діє; див. Додаток А для докладнішої інформації. Якщо ж нічого не відбулося або "P" виведено на консоль, то ця функція недоступна; ви зможете видаляти попередні символи лише один за одним.

Інтерпретатор діє подібно до оболонки Юнікса: якщо його викликано із стандартним виводом під'єднаним до терміналу, команди зчитуються і виконуються у діалоговому режимі, якщо ж його викликано з аргументом, що є назвою файла, або ж файл задано як стандартний ввід, інтерпретатор зчитує і виконує *скрипт* з цього файла.

Інший спосіб запуску інтерпретатора — це "python -c "*команда*" [аргумент] ...", що виконує дії, задані командою, подібно до того, як діє опція оболонки -c. Тому що інструкції Пайтона часто містять пробіли та інші спеціальні (з перспективи оболонки) символи, найкраще взяти *команду* в подвійні лапки.

Слід зауважити, що існує різниця між "python файл" і "python ‹ файл". В останьому випадку виклики таких функцій як input() і raw_input(), спробують зчитати ввід з файла. Оскільки парсер уже повністю зчитав файл ще до того, як почалося виконання програми, то сама програма відразу ж отримає кінець файла. У попередньому ж випадку (і це у більшості випадків саме те, що потрібно), ці функції отримають ввід з того файла чи пристрою, що під'єднаний до стандартного вводу інтерпретатора.

Коли використовується файл зі скриптом, інколи буває потрібно розпочати діалоговий режим по закінченні програми. Це можна зробити за допомогою опції -i перед скриптом. (З наведених вище причин цей метод не діє, коли скрипт зчитано зі стандартного вводу).

Передача аргументів

Якщо назва скрипту і додаткові аргументи відомі інтерпретатору, то вони можуть бути отримані всередині скрипту із системної змінної sys.argv, котра являє собою список рядків. Довжина списку — принаймні один елемент; якщо ж ні скрипту, ані аргументів не подано, sys.argv[0] є порожнім рядком. Якщо назва скрипта задана як '-'(що означає стандартний ввід), то відповідно і sys.argv[0] має значення '-'. Якщо використано ключ командного рядка -c, то sys.argv[0] отримує '-c'. Якщо використано -m модуль, sys.argv[0] отримує повну назву заданого модуля. Опції, задані після -c команда або -m модуль не використовуються інтерпретатором, а залишаються у sys.argv для опрацювання командою чи модулем.

Діалоговий режим

Коли команди зчитуються з терміналу, інтерпретатор працює у так званому діалоговому режимі. У цьому режимі кожен рядок починається головним запрошенням до вводу, здебільшого три знаки "більше" (">>>"); для продовження вводу рядок починається вторинним запрошенням до вводу, зазвичай три крапки ("... ").

Продовження вводу потрібно при вводі багаторядкової конструкції. Розгляньмо як приклад конструкцію з if:

```
>>>the_world_is_flat = 1
>>>if the_world_is_flat:
...     print "Дивись не впади!"
...
Дивись не впади!
```

Інтерпретатор і його середовище

Обробка помилок

Якщо трапилась помилка, інтерпретатор видає повідомлення про помилку разом з станом стеку. В діалоговому режимі він повертається до головного запрошення до вводу. Якщо дані було отримано з файла, інтерпретатор вийде з ненульовим статусом після того, як виведе дані стека. (Винятки, що контролюються оператором except у конструкції try у цьому контексті не вважаються помилками). Окремі помилки є безумовно фатальними і тому зумовлюють вихід програми з ненульовим статусом. Усі повідомлення про помилки виводяться на стандартний потік виводу помилок (stderr).

Нормальний вивід з команд, що виконуються, подається на стандартний вивід.

Якщо ввести символ переривання (зазвичай Control-C чи DEL) у відповідь на головне чи вторинне запрошення до вводу, весь ввід буде проігноровано й інтерпретатор повернеться до голового запрошення до вводу. (Цього може не статися через певну проблему із пакунком GNU Readline). Якщо ввести символ переривання під час виконання якоїсь команди, то утвориться виняток "переривання клавіатурою" (KeyboardInterrupt), який можна впіймати за допомогою конструкції try.

Виконання скриптів на мові Python

На Юнікс-подібних системах скрипти, написані на Пайтоні, можуть виконуватися безпосередньо, як скрипти оболонки, якщо додати рядок

```
#! /usr/bin/env python
```

(припускається, що за тією адресою встановлений інтерпретатор) на початку скрипта і задати виконуваний статус для файла. "#!" повинні бути першими двома символами файла. На окремих системах цей перший рядок повинен закінчитися юніксовим символом кінця рядка ("\n"), а не макінтошевим ("\r") чи віндовсним ("\r\n"). Зауважте, що символ "#" у мові Пайтон означає початок коментаря.

Файлу можна надати виконуваного статусу за допомогою команди chmod:

```
$ chmod +x myscript.py
```

Кодування джерела

Файл, що містить програму, написану на Пайтоні, може бути створений на основі кодування відмінного від ASCII. Найкращий спосіб вказати кодування — це додати спеціальний коментар відразу після рядка з "#!":

```
# -*- coding: кодування -*-
```

Завдяки цій декларації усі символи джерела будуть інтерпретовані як закодовані у вказаному *кодуванні*. Також це уможливлює задання значень в Юнікоді за допомогою зазначеного кодування. Список можливих кодувань можна знайти у Python Library Reference, під розділом codecs.

Якщо ваш текстовий редактор дозволяє зберігати файли у кодуванні UTF-8 з

притаманною йому позначкою порядку байтів (byte order mark або BOM), то ви можете використовувати її замість вищезгаданої декларації. Інтегроване середовище для розробки програм на Пайтоні (IDLE) має цю властивість, якщо задано Options/General/Default Source Encoding/UTF-8. (Опції/Загальне/Стандартне кодування джерела/UTF-8) Слід зауважити, що старіші версії Пайтона (2.2 і раніші) не розуміють цієї позначки, не розуміє її й операційна система у файлах з "#!".

Кодування UTF-8 (задане через підпис чи декларацію) дозволяє використовувати переважну більшість мов світу одночасно як у коментарях, так і для задання рядкових констант. Використання символів, що не належать до ASCII, у ідентифікаторах не дозволяється. Щоб відобразити ці символи належним чином ваш редактор повинен розпізнати, що файл закодовано в UTF-8, та використовувати шрифт, здатний відобразити всі символи файла.

Стартові файли для діалогового режиму

Коли ви використовуєте Пайтон у діалоговому режимі, часто потрібно виконувати певні стандартні команди кожного разу, коли включено інтерпретатор. Цього можна досягти шляхом задання змінної середовища PYTHONSTARTUP, яка вказує на файл з потрібними стартовими командами. Це схоже на використання файла .profile в оболонках Юнікса.

Цей файл зчитується лише в діалоговому сеансі, а не тоді, коли Пайтон виконує команди зі скрипту, чи коли /dev/tty явно задано як джерело усіх команд (в усьому іншому цей сеанс подібний до діалогового). Цей файл виконується у тому ж іменному просторі, де і всі інші діалогові команди. Таким чином, об'єкти, імпортовані та визначенні у ньому, можуть використовуватися безпосередньо під час діалогового сеансу. У цьому файлі також можливо змінити символи вводу sys.ps1 та sys.ps2.

Якщо потрібно зчитати додаткові стартові файли з поточного каталогу, то цього можна досягти за допомогою коду на зразок

```
if os.path.isfile('.pythonrc.py'):
    execfile('.pythonrc.py')
```

Якщо стартовий файл потрібно використати у скрипті, то ця дія повинна бути явно вказана:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
```

```
if filename and os.path.isfile(filename):
    execfile(filename)
```

Попередні зауваження

У поданих нижче прикладах ввід та вивід розрізняються за допомогою присутності чи відсутності запрошень до вводу (">>> " та "... "): щоб повторити приклад, слід ввести текст після того, як з'явиться запрошення. Рядки без запрошення - це вивід з інтерпретатора. Зауважте, що вторинне запрошення без жодного тексту означає, що треба ввести порожній рядок. Це використовується в багаторядкових командах.

Багато прикладів у цьому підручнику, включно з тими, які виконують в діалоговому режимі, містять коментарі. Коментарі у Пайтоні починаються символом "дієз" ("#") і продовжуються до кінця рядка. Коментар може з'явитися на початку рядка, після пробілу чи коду, але не всередині рядкової константи. Дієз всередині рядкової константи — звичайнісінький символ.

Окремі приклади:

```
# це перший коментар
SPAM = 1 # а це другий коментар
# ... а це третій!
STRING = "# А це — не коментар."
```

Python як калькулятор

Спробуймо декілька простих команд на Пайтоні. Запустіть інтерпретатор і дочекайтеся головного запрошення ">>> ".

Числа

Інтерпретатор діє як простий калькулятор: ви вводите вираз і отримуєте результат. Синтаксис математичних виразів досить традиційний: оператори +, -, *, / діють як і в більшості інших мов, наприклад, Pascal чи C. Дужки можуть використовуватися для групування. Наприклад:

```
>>> 2+2
4
>>> # Це — коментар
... 2+2
```

```
4
>>> 2+2 # і це коментар, на одному рядку з кодом
4
>>> (50-5*6)/4
5
>>>  # Ділення цілих чисел повертає нижню цілочисельну
величину (floor):
... 7/3
2
>>> 7/-3
-3
```

Подібно до C, знак рівності ("=") служить для присвоєння значення змінній. Результат присвоєння не виводиться:

```
>>> shyryna = 20
>>> vysota = 5*9
>>> shyryna * vysota
900
```

Значення може бути присвоєна кільком змінним одночасно:

```
>>> x = y = z = 0 # x, y та z дорівнюють нулю
>>> x
0
>>> y
0
>>> z
0
```

Повністю підтримуються числа з плаваючою крапкою; оператори зі змішаними типами компонентів перетворюють цілочислові компоненти на компоненти з плаваючою крапкою:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Комплексні числа також підтримуються. Уявні компоненти задаються суфіксом "j" чи "J". Комплексні числа з ненульовим реальним

компонентом пишуться як "`(реальне + уявнеj)`", або задаються через функцію "`complex(re,im)`".

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Комплексні числа завжди виражаються двома числами з плаваючою крапкою, реальною і уявною частинами. Щоб отримати ці частини з уявного числа z, слід використовувати `z.real` та `z.imag`.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Функції переведення у цілі числа та числа з плаваючою крапкою (`float()`, `int()` та `long()`) не працюють із комплексними числами, тому що не існує правильного методу перетворення комплексного числа в дійсне. Слід використовувати `abs(z)`, щоб отримати його абсоютну величину (у вигляді числа з плаваючою крапкою) чи `z.real` щоб отримати реальну частину.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
 File "stdin", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
```

```
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

У діаголовому режимі останній виведений вираз зберігається у змінній "_". При використанні Пайтона у якості настільного калькулятора ця риса дещо полегшує продовження обчислень:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
>>>
```

Цю змінну слід вважати такою, яку можна лише зчитувати. Явне присвоєння їй певного значення призведе до створення незалежної локальної змінної з такою самою назвою, що затьмарить вбудовану змінну із її чарівними властивостями.

Рядки

Окрім чисел Пайтон може працювати і з рядками, які можуть записуватись кількома способами. Вони можуть оточуватися одинарними чи подвійними лапками:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Текстові рядки, можуть займати кілька рядків. (примітка перекладу.

Каламбур multiline string (багаторядковий рядок) - це текстова константа, в якій міститься кілька символів переходу на новий рядок).

Якщо рядок закінчується символом "\", то наступний рядок є продовженням першого:

```
hello = "Це досить довгий рядок, що складається \n\
з кількох рядків тексту, подібно до того, як це робиться у С.\n\
 Зауважте, що додаткові пробіли на початку рядка теж рахуються."

print hello
```

Слід зазначити, що символи на позначення нового рядка (`\n`) повинні бути присутні; перехід на новий рядок, що йде за "\" не береться до уваги. Наведений вище приклад повинен вивести таке:

```
Це досить довгий рядок, що складається
з кількох рядків тексту, подібно до того, як це робиться у С.
 Зауважте, що додаткові пробіли на початку рядка теж рахуються.
```

Якщо створено "сирий" (raw) рядок, послідовність символів `\n` не замінюється символом переходу на новий рядок, а зберігається буквально. Таким чином цей приклад:

```
hello = r"Це досить довгий рядок, що складається \n\
з кількох рядків тексту, подібно до того, як це робиться у С."

print hello
```

виводить:

```
Це досить довгий рядок, що складається \n\
з кількох рядків тексту, подібно до того, як це робиться у С.
```

Також рядки можуть оточуватися трьома лапками: `"""` та `'''`.

Вживання символа "\" наприкінці рядка не потрібне, але він може бути всередині рядка:

```
print """
Використання: назва [ОПЦІЇ]
 -h                 Виводить це повідомлення
 -H   назва   сервера        Сервер,   до   якого   потрібно
підключитися
"""
```

Виводиться такий результат:

```
Використання: назва [ОПЦІЇ]
 -h                 Виводить це повідомлення
 -H   назва   сервера        Сервер,   до   якого   потрібно
підключитися
```

Інтерпретатор виводить рядок точнісінько так само, як його було задано всередині лапок, при цьому всі спеціальні символи можуть задаватися за допомогою контрольних послідовностей, що починаються зі зворотньої косої риски (\). Рядок береться в подвійні лапки, якщо всередині є одинарні лапки і немає подвійних, інакше — в одинарні лапки. (Оператор `print`, описаний далі, може використовуватися для виводу рядків без лапок чи контрольних послідовностей).

Конкатенація (з'єднання) рядків робиться за допомогою оператора `+`, а повторення — за допомогою `*`:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '&lt;' + word*5 + '&gt;'
'&lt;HelpAHelpAHelpAHelpAHelpA&gt;'
```

Дві символьні константи, виражені рядками, автоматично з'єднуються, якщо вони розташовані поруч. У наведеному вище прикладі перший рядок може бути написаний як "`word = 'Help' 'A'`". Це працює лише з двома символьними константами, а не з будь-якими виразами, що повертають як результат рядки:

```
>>> 'str' 'ing' # &lt;- Гаразд
```

```
'string'
>>> 'str'.strip() + 'ing' # &lt;- Гаразд
'string'
>>> 'str'.strip() 'ing' # &lt;- А це не годиться
 File "&lt;stdin&gt;", line 1, in ?
 'str'.strip() 'ing'
 ^
SyntaxError: invalid syntax
```

Рядки можуть індексуватись. Як і в C перший символ рядка має індекс 0. Окремого символьного типу не існує; символ — це просто рядок довжиною в один символ. Подібно до мови Icon, частини рядків можуть позначатися за допомогою двох індексів, розділених двокрапкою. Така операція називається **зріз**.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Пропуск першого індексу рівнозначний нулю, а пропуск останього - довжині рядка.

```
>>> word[:2] # Перші два символи
'He'
>>> word[2:] # Усі символи, крім перших двох
'lpA'
```

На відміну від C, рядки у Пайтоні не можуть бути змінені. Присвоєння значення індексованій позиції рядка призводить до помилки:

```
>>> word[0] = 'x'
Traceback (most recent call last):
 File "&lt;stdin&gt;", line 1, in ?
TypeError: object doesn't support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
 File "&lt;stdin&gt;", line 1, in ?
TypeError: object doesn't support slice assignment
```

Натомість створення нового рядка за допомогою індексованих елементів швидке й ефективне:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Ось корисна інваріантна операція, де `s[:i] + s[i:]` дорівнює `s`:

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Неправильна індексація частин рядка не призводить до помилки при зрізах: завеликий індекс замінюється на довжину рядка, а якщо верхній індекс більший за нижній, то утворюється порожній рядок:

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Якщо індекси від'ємні, відлік починається з кінця рядка. Наприклад:

```
>>> word[-1] # Останній символ
'A'
>>> word[-2] # Передостанній символ
'p'
>>> word[-2:] # Два останні символи
'pA'
>>> word[:-2] # Усі, окрім двох останніх символів
'Hel'
```

Зауважте, що -0 це те саме, що й 0:

```
>>> word[-0] # ( -0 дорівнює 0)
```

```
'H'
```

Завеликі від'ємні індекси автоматично скорочуються у зрізах, але це не працює для одинарних індексів (для окремих символів):

```
>>> word[-100:]
'HelpA'
>>> word[-10] # помилка
Traceback (most recent call last):
 File "&lt;stdin&gt;", line 1, in ?
IndexError: string index out of range
```

Найкращий спосіб запам'ятати, як працює індексація рядків, це уявити, що індекси вказують на позицію *між* символами, де лівий край має індекс 0. Тоді правий край рядка довжиною в *n* символів має індекс *n*, наприклад:

H	e	l	p	A
0	1	2	3	4

Перший ряд чисел дає позиції індексів від 0 до 5, а другий - відповідні від'ємні індекси. Частина рядка від *i* до *j* складається з усіх символів, розташованих між краями, що позначені як *i* та *j*.

Для додатніх індексів, довжина частини рядка дорівнює різниці індексів (якщо обидва знаходяться в допустимих межах). Наприклад, довжина `word[1:3]` дорівнює 2.

Вмонтована функція `len()` повертає довжину рядка:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Рядки у кодуванні Unicode

Починаючи з версії 2.0, Пайтон має новий тип даних для зберігання тексту: юнікодовий об'єкт. Він може використовуватися для зберігання

та обробки даних у Unicode (див. [1]) і добре інтегрується з існуючими рядковими об'єктами та здійснює автоматичну конверсію за потребою.

Перевага кодування Unicode полягає в тому, що воно визначає єдину ординальну величину для кожного символа у будь-якій письмовій системі сучасних чи давніх мов. До цього існувало лише 256 ординальних величин для позначення символів і текст здебільшого прив'язувався до кодування, що поєднувало ординальні величини з символами алфавіту. Це призводило до численних непорозумінь, особливо при інтернаціоналізації (цей термін традиційно позначається як "`i18n`" -- "i" + 18 символів посередині + "n") програмного забезпечення. Ця проблема вирішена в кодуванні Unicode, де одна кодова сторінка описує всі скрипти.

Створення юнікодових рядків у Пайтоні таке ж просте, як і створення звичайних рядків:

```
>>> u'Hello World !'
u'Hello World !'
```

Маленький символ "u" перед лапками означає, що задається юнікодовий рядок. Задання спеціальних символів у рядку може бути зроблено за допомогою юнікодових контрольних послідовностей (Unicode-Escape encoding) мови Пайтон, як це показано у наступному прикладі:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

Контрольна послідовність `\u0020` вказує, що у заданій позиції повинен бути вставлений символ, що має ординальну величину 0x0020 (пробіл).

Інші символи інтерпретуються через пряме використання їхніх ординальних величин як ординальних величин кодування Unicode. Якщо ваші буквальні величини задано за допомогою кодування Latin-1, що використовується у багатьох західних країнах, то для вас перші 256 символів кодування Unicode ті самі, що й 256 символів кодування Latin-1.

Подібно до звичайних рядків, юнікодові рядки можуть задаватися у

сирому режимі. Для цього слід додати префікс 'ur' перед відкриттям лапок для того, щоб Пайтон використовував "сире" кодуваня юнікодових контрольних послідовностей (Raw-Unicode-Escape encoding). При цьому переведення послідовностей `\uXXXX` у юнікодові символи відбудеться лише тоді, коли маленькій літері 'u' передує непарна кількість зворотніх скісних рисок (\).

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\\u0020World !'
```

Сирий режим найкорисніший, коли потрібно ввести багато зворотніх скісних рисок, що може бути необхідно у регулярних виразах.

Окрім цих стандартних кодувань, Пайтон має багато інших способів для створення юнікодових рядків на основі певного відомого кодування.

Вбудована функція `unicode()` надає доступ до всіх зареєстрованих юнікодових кодеків (codec < "COders and DECoders"). Серед найвідоміших кодувань, що можуть конвертуватися цими кодеками - Latin-1, ASCII, UTF-8, та UTF-16. Останні два — кодування змінної довжини, що зберігають юнікодові символи в одному чи більше байтах. Типове кодування — це здебільшого ASCII, що дозволяє лише симлоли від 0 до 127 і видає помилку, коли знаходить інші символи. Коли юнікодовий рядок виводиться на стандартний вивід, записується у файл чи конвертується за допомогою `str()`, то конверсія відбувається за допомогою цього стандартного кодування.

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"&auml;&ouml;&uuml;"
u'\xe4\xf6\xfc'
>>> str(u"&auml;&ouml;&uuml;")
Traceback (most recent call last):
 File "&lt;stdin&gt;", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode
characters in position 0-2: ordinal not in range(128)
```

Для конверсії юнікодових рядків у восьмибітові за допомогою певного кодування, юнікодові об'єкти мають спеціальний метод `encode()`, що отримує назву кодування як аргумент. Бажано вживати маленькі літери на позначення кодування.

```
>>> u"&auml;&ouml;&uuml;".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

Якщо ви маєте дані у певному кодуванні і хочете утворити з них відповідний юнікодовий рядок, то можете використовувати функцію `unicode()`, чий другий аргумент є назвою кодування.

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

Списки

Пайтон знає кілька *складних* типів даних, що використовуються для групування значень. Найбільш універсальним є *список*, що може бути створений як послідовність елементів, розділених комами і оточених прямими дужками. Елементи списку не обов'язково повинні належати одному типу.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Подібно до рядків ідекси списків починаються з 0. Списки можуть поділятися на частини, з'єднуватися тощо:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam',
'eggs', 100, 'Boe!']
```

На відміну від рядків, що є *незмінними*, індивідуальний елемент списку може бути змінено:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Призначення нових елементів частині списку можливе. Ця операція може навіть змінити розмір списку:

```
>>> # Замінити певні елементи:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Видалити елементи:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Додати:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> a[:0] = a # Вставити (копію) самого себе на початку
```

```
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy',
1234]
```

Стандартна функція `len()` також застосовується до списків:

```
>>> len(a)
8
```

Можливо також створити вкладені списки (тобто списки, що містять інші списки), наприклад:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra') # Див. розділ 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Слід зауважити, що в останьому прикладі, `p[1]` і `q` посилаються та той самий об'єкт! Ми повернемося до *об'єктної семантики* пізніше.

Перші кроки до програмування

Звичайно, Пайтон може використовуватися і для вирішення складніших задач, аніж додавання двох до двох. Зокрема, ми могли б написати початок послідовності *чисел Фібоначчі* таким чином:

```
>>> # числа Фібоначчі:
... # сума двох елементів визначає наступний:
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
```

```
...
1
1
2
3
5
8
```

Цей приклад ілюструє кілька нових властивостей:

- У першому рядку є численне присвоєння (multiple assignment): змінні `a` та `b` одночасно отримують нові значення 0 та 1. В останьому рядку ми знову зустрічаємо подібне присвоєння, при цьому вирази, розташовані по праву сторону знаку присвоєння, обчислюються перед тим як відбувається присвоєння. Правосторонні вирази обчислюються зліва направо.
- Цикл `while` виконується поки умова (тут: `b < 10`) залишається істинною. У Пайтоні, як і в C, будь-яка ненульове цілочисельне значення є істинним, а нульове — хибним. Умова може також бути рядком або списком, та й взагалі будь-якою послідовністю. Будь-яке значення з ненульовою довжиною — істинне, а пуста послідовність — хибна. Умова, використана у цьому прикладі, — просте порівняння. Стандартні оператори порівняння пишуться так само, як і в C: `<` (менше ніж), `>` (більше ніж), `==` (дорівнює), `<=` (менше ніж або дорівнює), `>=` (більше ніж або дорівнює) і `!=` (не дорівнює).
- *Тіло* циклу *виділяється відступами*: у цей спосіб Пайтон групує інструкції. Наразі Пайтон (ще![1]) не має розумної системи для редагування вводу, отже табуляції чи пробіли потрібно вводити "вручну" для кожного виділеного рядка. На практиці складніші програми на Пайтоні створюються за допомогою текстового редактора і більшість текстових редакторів має певний механізм для автоматичного виділення тексту. При вводі блоку інструкцій у діалоговому режимі він повинен закінчуватись пустим рядком, щоб позначає завершення (тому що інакше парсер не може вгадати, коли було введено останій рядок). Слід зауважити, що всередині простого блоку, всі рядки повинні бути виділені однаковою кількістю пробілів (чи табуляцій).

- Оператор `print` виводить значення переданих йому виразів. Рядки виводяться без лапок і пробіл вставляється між окремими елементами, як показано тут:

```
>>> i = 256*256
>>> print 'Значення і складає', i
Значення і складає 65536
```

Кінцева кома означає, що вивід не закінчується новим рядком:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Зауважте, що інтерпретатор додає символ нового рядка перед тим, як вивести запрошення до вводу, якщо останній рядок не було закрито.

Окрім оператора `while`, про який щойно йшлося, Пайтон знає звичайні контрольні структури, що застосовуються і в інших мовах програмування (хоча і з певними "відхиленнями").

Оператор `if`

Чи не найвідомішим твердженням є `if`. Приклад:

```
>>> x = int(raw_input("Прошу ввести ціле число: "))
>>> if x < 0:
...     x = 0
...     print "Від'ємне число замінено на нуль"
... elif x == 0:
...     print 'Нуль'
... elif x == 1:
```

```
...     print 'Один'
... else:
...     print 'Більше'
...
```

Ця конструкція може мати нуль або більше частин `elif`, частина `else` — необов'язкова. Ключове слово `elif` — це скорочення від "else if" ("інакше якщо"), його стисла форма запобігає надмірному виділенню пробілами. Послідовність `if` ... `elif` ... `elif` ... є відповідником операторів `switch` та `case`, що зустрічаються в інших мовах програмування.

Оператор `for`

Оператор `for` у Пайтоні трохи відрізняється від того, до якого ви могли звикнути, використовуючи C або Pascal. Замість постійного перебору чисел арифметичної прогресії (як у Pascal) чи надання користувачеві можливості визначати як крок перебору (ітерації), так і кінцеву умову (як у C), оператор `for` у мові Пайтон перебирає члени будь-якої послідовності (списку чи рядка) у порядку їхнього в ній розташування. Наприклад:

```
>>> # Довжини рядків:
... a = ['кіт', 'вікно', 'жбурляти']
>>> for x in a:
...     print x, len(x)
...
кіт 3
вікно 5
жбурляти 8
```

Модифікація членів послідовності під час перебору небезпечна (це, власне, може трапитися лише зі змінюваними типами, як, скажімо, списки). Якщо потріно змінити список, що в даний час перебирається, (наприклад, подвоїти певні члени), то для цього слід перебирати копію списку. Синтаксис зрізів дозволяє це зручно робити:

```
>>> for x in a[:]: # зробити копію цілого списку за
допомогою зрізів
...     if len(x) &gt; 6: a.insert(0, x)
...
```

```
>>> a
['жбурляти', 'кіт', 'вікно', 'жбурляти']
```

Функція `range()`

Якщо потрібно перебрати послідовність чисел, то тут стане в нагоді стандартна функція `range()`. Вона створює списки, що містять арифметичні прогресії:

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

Задане останнє значення ніколи не є частиною створеного списку: `range(10)` створює список із десяти елементів, які відповідають індексам послідовності довжиною 10 елементів. Можливо також задати початок списку іншим числом та вказати інший крок (навіть від'ємний):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Щоб перебрати індекси послідовності, слід використовувати функції `range()` та `len()` таким чином:

```
>>> a = ['У', 'Марічки', 'є', 'ягнятко']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 У
1 Марічки
2 є
3 ягнятко
```

Оператор `break` та `continue`; конструкція `else` у циклах

Оператор `break`, як і в C, перериває найближчий цикл `for` чи `while`.

Твердження `continue`, також запозичене з C, продовжує перебір з наступного кроку.

Циклічні оператори можуть також мати конструкцію `else`, яка виконується, коли цикл завершується виснаженням списку (для `for`) або коли умова перестає бути істинною (для `while`), але не тоді, коли цикл закінчується примусово, за допомогою `break`. Наведений нижче код показує, як це діє на прикладі пошуку простих чисел:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'дорівнює', x, '*', n/x
...             break
...     else:
...     # ми потрапили сюди, бо не знайшли спільного
множника
...     print n, 'просте число'
...
2 просте число
3 просте число
4 дорівнює 2 * 2
5 просте число
6 дорівнює 2 * 3
7 просте число
8 дорівнює 2 * 4
9 дорівнює 3 * 3
```

Оператор `pass`

Оператор `pass` не робить нічого. Він використовується тоді, коли хоч якась інструкція необхідна синтаксично, але програма не потребує жодної дії. Наприклад:

```
>>> while True:
...     pass # Очікування сигналу переривання з
клавіатури
...
```

Визначення функцій

Створімо функцію, що виводить числа Фібоначчі до певної межі:

```
>>> def fib(n): # вивести числа Фібоначчі до n
...     """Вивести числа Фібоначчі до n."""
...     a, b = 0, 1
...     while b &lt; n:
...         print b,
...         a, b = b, a+b
...
>>> # Тепер викликаємо щойно задану функцію:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Ключове слово `def` вводить *визначення* (definition) функції. За ним повинна бути назва функції та оточений дужками список формальних параметрів. Інструкції які утворюють тіло функції починаються з наступного рядка і повинні бути виділені пробілами. Першим, але необов'язковим, рядком функції може бути символьна константа, яка коротко описує функцію. Її називають рядком документації.

Існують спеціальні утиліти, що використовують документаційні рядки для автоматичного створення друкованої чи онлайн документації або для перегляду коду в діалоговому режимі. Документування коду — дуже гарна звичка, отож спробуйте не забувати про це.

Виконання функції вводить новий простір імен, що використовується для локальних змінних функції. Зокрема, всі присвоєння змінним всередині функції зберігають свої значення у локальному просторі імен, тоді як при посиланні на змінну пошук починається у локальному, а потім продовжується у глобальному, і наприкінці — у просторі імен вбудованих ідентифікаторів. Таким чином, глобальні змінні не можуть отримувати нові значення всередині функцій (за винятком якщо вони названі у твердженні `global`), хоча посилання на них можливе.

Параметри (або аргументи) функції вводяться в простір імен функції при її виклику. Аргументи передаються за значенням, де *значення* - завжди *посилання* на об'єкт, а не значення самого об'єкта, тому точніше було б сказати - передача за посиланням. При передачі змінюваного об'єкта будь-які його зміни всередині викликаної функції стануть видимі в середовищі, що викликало цю функцію, наприклад, додання нових елементів до списку. Коли одна функція викликає іншу, то створюється новий локальний простір імен для цього виклику.

Визначення функції додає назву функції до поточного простору імен. Значення назви функції належить до типу, який ідентифікується інтерпретатором як задана користувачем функція. Це значення може присвоюватися іншій змінній, що потім теж може використовуватися як функція. Тут показано, як діє загальний механізм перейменування:

```
>>> fib
&lt;function object at 10042ed0&gt;
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Можливо хтось скаже що `fib` — не функція, а процедура. У Пайтоні, як і в С, процедури — це функції, що не повертають жодного значення. Власне, з технічної точки зору, процедури таки повертають певне значення, хоча й досить нецікаве. Це значення `None`. Зазвичай, це значення не виводиться інтерпретатором, якщо це єдине можливе значення для виводу. Але якщо дійсно хочеться його побачити, його потрібно роздрукувати явно:

```
>>> print fib(0)
None
```

Створення функції, що повертає список чисел Фібоначчі замість виведення їх на друк, досить просте:

```
>>> def fib2(n): # повертає числа Фібоначчі до n
...     """Повертає список чисел Фібоначчі до n"""
...     result = []
...     a, b = 0, 1
...     while b &lt; n:
...         result.append(b) # див. нижче
```

```
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100) # виклик функції
>>> f100 # вивід результату
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Цей приклад також демонструє кілька нових властивостей мови Пайтон:

- Оператор `return` повертає із функції певне значення. Якщо `return` вжито без аргументів, то результатом повернення є `None`. Якщо процедура закінчується сама по собі, то результатом повернення є `None`.
- Інструкція `result.append(b)` викликає метод об'єкта списку `result`. Метод — це функція, що "належить" певному об'єкту. Вона викликається у формі `obj.назва_методу`, де `obj` — певний об'єкт (може також бути виразом) і `назва_методу` — назва методу, визначеного типом об'єкта. Різні типи визначають різні методи. Методи різних типів можуть мати однакову назву, не спричиняючи при цьому двозначності. (Ви можете також визначити і ваші власні методи за допомогою *класів*, про що йтиметься далі). Наведений у цьому прикладі метод `append()` визначений для спискових об'єктів; він додає новий елемент в кінець списку. У цьому прикладі це еквівалентно "`result = result + [b]`", але цей метод є більш ефективним.

Докладніше про визначення функцій

Можливо також визначити функцію зі змінною кількістю аргументів. Для цього існує три способи, що можуть сполучуватися.

Стандартні значення аргументів

Найкорисніший спосіб — це визначити типове значення для одного чи кількох аргументів. Це створює можливість виклику функції з меншою кількістю аргументів, аніж задано у визначенні функції. Наприклад:

```
def ask_ok(prompt, retries=4, complaint='Так чи ні, будь-ласка!'):
    while True:
        ok = raw_input(prompt)
```

```
        if ok in ('т', 'та', 'так'): return True
        if ok in ('н', 'ні', 'нєт'): return False
        retries = retries — 1
        if retries &lt; 0: raise IOError, 'затятий
користувач'
        print complaint
```

Ця функція може викликатися як `ask_ok('Справді закрити програму?')` чи як `ask_ok('Переписати файли?', 2)`.

Цей приклад також ілюструє ключове слово `in`, що дозволяє перевірити чи дана послідовність містить певний елемент.

Стандартні значення обчислюються в момент задання функції відповідно до *визначаючого* контенсту, тому:

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

виведе `5`.

Важливе зауваження: стандартнне значення обчислюється лише раз. Хоча існує виняток, коли стандартнне значення — змінюваний об'єкт, скажімо, список, словник, реалізація більшості класів. Наприклад, наступна функція акумулює аргументи, що передаються при подальших викликах:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

Це виведе:

```
[1]
[1, 2]
[1, 2, 3]
```

Якщо ви не хочете, щоб стандартне значення було спільним для всіх наступних викликів, то можна створити функцію на зразок:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Ключові аргументи

Функції можуть також викликатися за допомогою *ключових аргументів* у вигляді "*ключ = значення*". Наприклад, ця функція:

```
def parrot(voltage, state='a stiff', action='voom',
type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "Volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

може викликатися у будь-який вказаний нижче спосіб:

```
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

але такі виклики неправильні:

```
parrot() # пропущено обов'язковий аргумент
parrot(voltage=5.0, 'dead') # неключовий аргумент
передається після ключового
parrot(110, voltage=220) # два значення для одного
аргумента
```

```
parrot(actor='John Cleese') # невідомий ключ
```

Взагалі в списку аргументів позиційні аргументи повинні бути розташовані перед ключовими, при цьому ключі повинні бути вибрані з формальних назв параметрів. Чи задані для цього параметра стандартні значення — не важливо. Жоден аргумент не може отримувати значення більш, ніж один раз — формальні назви параметрів, що відповідають позиційним аргументам не можуть використовуватися як ключові слова під час того самого виклику. Ось приклад того, коли помилка відбувається саме через це обмеження:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
 File "&lt;stdin&gt;", line 1, in ?
TypeError: function() got multiple values for keyword
argument 'a'
```

Якщо останній формальний параметр задано у формі ***назва*, то він отримує *словник*, що складається з аргументів, чиї ключі відповідають формальним параметрам. Він може сполучатися з формальним параметром у формі **назва* (описаний в наступному підрозділі), який отримує кортеж, що складається з позиційних аргументів, не включених у список формальних пареметрів (аргумент **назва* повинен передувати аргументу ***назва*). Наприклад, функцію, задану таким чином:

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, '?'
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments: print arg
    print '-'*40
    keys = keywords.keys()
    keys.sort()
    for kw in keys: print kw, ':', keywords[kw]
```

можна викликати отак:

```
cheeseshop('Limburger', "It's very runny, sir.",
        "It's really very, VERY runny, sir.",
```

```
        client='John Cleese',
        shopkeeper='Michael Palin',
        sketch='Cheese Shop Sketch')
```

Що, звичайно, виведе:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Зауважте, що метод ключових аргументів `sort()` викликано перед виведенням змісту словника `keywords`, бо інакше порядок виведення аргументів був би невизначеним.

Списки аргументів довільної довжини

Нарешті, остання часто використовувана можливість — визначення функції, що може бути викликана з будь-якою кількістю аргументів. Ці аргументи передаються за допомогою кортежа. Нуль чи більше звичайних аргументів можуть передувати змінній кількості аргументів.

```
def fprintf(file, format, *args):
    file.write(format % args)
```

Розпакування списків аргументів

Зворотня ситуація трапляється, коли аргументи задані списком чи кортежем, але їх потрібно розпакувати для виклику функції, що потребує окремих позиційних аргументів. Наприклад, вбудована функція `range()` потребує двох окремих аргументів, що вказують на межі послідовності. Якщо вони не задані окремо, виклик функції слід писати з оператором `*`, що дозволяє розпакувати аргументи, задані списком чи кортежем:

```
>>> range(3, 6) # звичайний виклик з окремими аргументами
[3, 4, 5]
>>> args = [3, 6]
```

```
>>> range(*args) # виклик із аргументами, розпакованими зі списку
[3, 4, 5]
```

Лямбда-функці

За популярною вимогою до Пайтона було додано кілька нових властивостей, типових для функціональних мов програмування та мови Lisp. Ключове слово `lambda` дозволяє створювати невеличкі анонімні функції. Ось, наприклад, функція, що повертає суму двох своїх аргументів: "`lambda a, b: a+b`". Лямбда-функції можуть стати в нагоді, коли потрібні об'єкти функцій. Синтаксично вони обмежені одним єдиним виразом. Семантично вони просто синтаксичний цукор нормального визначення функції. Подібно до вкладених функцій лямбда-функції можуть посилатися на змінні із зовнішнього контексту:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Рядки документації

Сформована певна домовленість щодо форматування документації.

Перший радок повинен містити стислу інформацію про об'єкт. Заради стислості в ньому не повинно бути назви чи типу об'єкта, бо їх можна отримати і в інший спосіб (за винятком коли назва є дієсловом, що описує дію функції). Цей рядок повинен починатися з великої літери і закінчуватися крапкою.

Якщо текст документації складається з кількох рядків, то другий рядок повинен бути пустим, що візуально відокремлює заголовок від решти опису. Наступні рядки повинні складатися з одного чи більше абзаців, що описують ньюанси виклику об'єкта, побічні ефекти тощо.

Інтерпретатор Пайтона не відкидає пропуски на початку рядка у багаторядкових символьних константах, виражених рядками, отже утиліти, що обробляють документацію, при потребі повинні відкидати ці

пробіли. Перший непустий рякок після заголовка визначає кількість пробілів для всього подальшого тексту документації. (Ми не можемо використовувати для цієї мети перший рядок, тому що він загалом розташований відразу після початкових лапок, а отже не виділений пробілами). Кількість початкових пробілів, "еквівалентна" знайденій у цьому рядку потім видаляється від початку усіх наступних рядків. Рядки з меншою кількістю пробілів не повинні зустрічатися, але якщо таки зустрічаються, то всі початкові пробіли повинні видалятися. Еквівалентність пропусків повинна перевірятися після розгортання табуляцій (зазвичай у вісім пробілів).

Ось приклад багаторядкової документації:

```
>>> def my_function():
...     """Не робить нічого, лише містить документацію.
...
...     Справді, ця функція не робить нічого.
...     """
...     pass
...
>>> print my_function.__doc__
Не робить нічого, лише містить документацію.

    Справді, ця функція не робить нічого.
```

Структури даних

У цьому розділі поглиблено розглядаються речі, про які ви вже дещо довідалися, а також кілька нових тем.

Докладніше про списки

Спискові типи даних мають і інші методи. Нижче подано всі методи спискових об'єктів:

`append(x)`

Додає новий елемент в кінець списку; еквівалентно виразу `a[len(a):] = [x]`.

`extend(L)`

Розширює список шляхом додання всіх елементів до даного списку; еквівалентно виразу `a[len(a):] = L`.

`insert(i, x)`

Вставити елемент у заданій позиції. Перший аргумент — індекс елемента, перед яким потрібно зробити вставку, таким чином `a.insert(0, x)` додає новий елемент на самому початку списку, а `a.insert(len(a), x)` еквівалентно виразу `a.append(x)`.

`remove(x)`

Видаляє перший елемент списку зі значенням x. Видалення неіснуючого елемента є помилкою.

`pop([i])`

Видаляє елемент із заданої позиції та повертає його. Якщо індекс не вказано, `a.pop()` видадяє і повертає останній елемент списку. (Квадратні дужки навколо `i` у прототипі методу вказують на те, що параметр не є обов'язковим, а не на те, що квадратні дужки потрібно ввести у вказаній позиції. Ця нотація доволі часто зустрічається у "Довіднику з мови Python".

`index(x)`

Повертає індекс першого елемента зі значенням x. Посилання на неіснуючий індекс є помилкою.

`count(x)`

Повертає кількість елементів x у списку.

`sort()`

Впорядковує елементи списку (на місці).

`reverse()`

Організовує елементи в зворотньому порядку (на місці).

Ось приклад, що використовує більшість спискових методів:

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
```

```
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

Використання списку як стека (магазина)

Спискові методи дозволяють у доволі простий спосіб використання списків як стека, де перший елемент, що додається, першим і знімається ("останнім ввійшов, першим вийшов"). Щоб додати новий елемент на вершину стека, слід використовувати метод `append()`. Щоб видалити елемент з вершини стека, слід використовувати метод `pop()` без задання індексу. Наприклад:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Використання списку як черги

Списки також зручно використовувати як черги (queue), де перший елемент, що додано, знімається першим ("першим ввійшов, першим вийшов"). Щоб додати елемент в кінець черги, використовуйте `append()`. Щоб видалити елемент з початку черги — використовуйте `pop()` з індексом 0. Наприклад:

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry") # Terry arrives
>>> queue.append("Graham") # Graham arrives
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

Засоби функціонального програмування

Існують кілька надзвичайно корисних вбудованих функцій для роботи зі списками: `filter()`, `map()`, та `reduce()`.

"`filter(функція, послідовність)`" повертає послідовність (того самого типу, якщо можливо), що складається з тих елементів послідовності, для яких *функція(елемент)* є істинною.

Наприклад, щоб віднайти прості числа:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

"`map(функція, послідовність)`"
викликає `функцію(елемент)` для кожного елемента послідовності і повертає список значень, повернутих функцією. Наприклад, щоб обчислити куби:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

`map` може отримвувати більше ніж одну послідовість; при цьому кількість аргументів функції повинна відповідати кількості послідовностей, а сама функція викликається з відповідним елементом кожної послідовності (або з `None`, якщо одна послідовність коротша за іншу). Наприклад:

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

"`reduce(функція, послідовність)`" повертає єдине значення, що утворюється шляхом виклику бінарної *функції* для перших двох елементів послідовності, потім для результату і наступного елемента і т. д. Наприклад, щоб обчислити суму чисел від 1 до 10:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

Якщо послідовність має лише один елемент, повертається його значення, якщо ж послідовність пуста — викликається виняток.

Може також передаватися і третій елемент, що задає початкове значення. У цьому випадку початкове значення повертається якщо послідовність — пуста. Сама ж функція застосовується до початкового значення і першого елемента, потім — до результату і другого і т.д. Наприклад:

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

Приклад функції `sum()`, поданий у цьому прикладі використовувати не слід: обчислення суми є настільки поширеною операцією, що для цього існує вбудована функція `sum(послідовність)` і діє вона так само, як і подана вище функція. Цю функцію було додано у версії 2.3.

Включення списків

Включення списків (list comprehension) надає можливість створювати списки без застосування функцій `map()`, `filter()` та `lambda`. Визначення списку, що утворюється, часто є набагато "чистішим", ніж створення списків за допомогою зазначених функцій. Кожне включення списку складається з виразу, за яким іде конструкція `for`, а потім — нуль чи більше конструкцій `for` чи `if`. Список, що утворюється, є результатом обчислення виразу в контексті розташованих за ним конструкцій `for` та `if`. Якщо обчислення виразу видає кортеж, то вираз повинен бути оточеним дужками.

```
>>> freshfruit = [' banana', ' loganberry ', 'passion
fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x &gt; 3]
[12, 18]
>>> [3*x for x in vec if x &lt; 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec] # помилка — навколо
кортежа потрібні дужки
 File "&lt;stdin&gt;", line 1, in ?
 [x, x**2 for x in vec]
             ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
```

```
[8, 12, -54]
```

Включення списків є набагато гнучкішим за `map()` і може застосовуватися до функцій, що отримують більш ніж один аргумент, а також до вкладених функцій (nested functions):

```
>>> [str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Оператор `del`

Видалити елемент з даного списку через посилання на його індекс можна за допомогою оператора `del`. Він також може використовуватися для видалення частин списку (що було зроблено раніше шляхом призначення пустого списку частинам, призначеним для видалення). Наприклад:

```
>>> a = [-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

`del` може також використовуватися для видалення змінних:

```
>>> del a
```

Посилання на назву `a` після цього є помилкою (якщо тільки вона не отримує нове значення (перевизначиться)). Далі йтиметься і про інші застосування `del`.

Кортежі та послідовності

Ми бачили, що списки і рядки мають чимало спільних властивостей, зокрема індексацію та операції зрізів (slicing operations). Вони є прикладами типів даних *послідовностей*. Оскільки Пайтон є мовою, що еволюціонує, інші типи даних, що належать до послідовностей, можуть додатися в майбутньому. Існує також інший тип даних, що належить до послідовностей, — *кортеж*.

Кортеж складається з певної кількості елементів, розділених комами, наприклад:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Кортежі можуть також вкладуватися:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Як ви бачите, при виводі кортежі завжди оточуються дужками, що допомагає правильно інтерпретувати вкладені кортежі. При вводі вони також можуть оточуватися дужками, хоча і не обов'язково. У більшості ж випадків дужки потрібні у будь-якому разі (якщо кортеж є частиною більшого виразу).

Кортежі мають багато застосувань. Наприклад: пара координат (x, y), запис із бази даних тощо. Кортежі, подібно до рядків, є незмінними: присвоєння нового значення певному елементу кортежа - неможливе (втім, це можна симулювати за допомогою зрізів та конкатенації. Можливо також створити кортежі, що містять змінні об'єкти, такі як списки.

Створення кортежів довжиною 0 чи 1 представляє певну проблему, але для цього існують певні синтаксичні особливості. Пусті кортежі

утворюються за допомогою пустої пари дужок. Одноелементний кортеж утворюється за допомогою певного елемента та коми (оточення елемента дужками — не є достатнім, бо дужки використовуються і в виразах). Негарно, зате ефективно. Наприклад:

```
>>> empty = ()
>>> singleton = 'hello', # &lt;-- зауважте кінцеву кому
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Інструкція `t = 12345, 54321, 'hello!'` є прикладом *пакування кортежа*: значення `12345`, `54321` та `'hello!'` "запаковані" всередиі кортежа. Зворотня операція також можлива:

```
>>> x, y, z = t
```

Це зветься (досить доречно) *розпаковуванням послідовностей*. Розпаковування послідовностей вимагає, щоб кількість змінних по ліву сторону дорівнювала кількості елементів у послідовності. Зауважте, що чисельне присвоєння (multiple assignment) є лише комбінацією пакування кортежів та розпакування послідовностей!

При цьому існує певна асиментія: пакування кількох значень завжди створює кортеж, а розпаковування — працює з довільними послідовностями.

Множини

Пайтон також має спеціальний тип даних для *множин*. Множина (set) — це невпорядкована сукупність неповторюваних елементів.

Серед основних застосувань множин є такі як перевірка наявності елемента чи видалення повторюваних елементів. Об'єкти, що виражають множини включають такі математичні операції як об'єднання (union), перетин (intersection), різниця (difference), та симетрична різниця (symmetric difference).

Ось кілька коротких прикладів:

```
>>> basket = ['apple', 'orange', 'apple', 'pear',
'orange', 'banana']
>>> fruits = set(basket) # створити множину без
повторів
>>> fruits
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruits # швидка перевірка наявності
елемента
True
>>> 'crabgrass' in fruits
False

>>> # Ілюструє операції над неповторюваними літерами
з двох слів
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a # кількісь неповторюваних літер в a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b # різниця (літери, що є в a, але не в b)
set(['r', 'd', 'b'])
>>> a | b # об'єднання (літери, що є або в a або в b)
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b # перетин (літери, водночас присутні в a та
b)
set(['a', 'c'])
>>> a ^ b # симетрична різниця (літери, присутні або
в a або в b, але не в обох одночасно)
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Словники

Словник — це ще один корисний тип даних мови Пайтон. Інколи словники можна зустріти в інших мовах під назвою "асоціативна пам'ять" чи "асоціативні масиви". На відміну від послідовностей, що індексуються за допомогою чисел, словники індексуються за допомогою *ключів*, які можуть належати до будь-якого незмінного типу; рядки і числа завжди можуть бути ключами. Кортежі також можуть бути ключами, якщо вони складаються лише з рядків, чисел чи кортежів. Якщо ж кортеж містить, прямо чи опосередковано, певний змінний об'єкт, то він не може використовуватися як ключ. Використання списків у якості ключів неможливе, тому що списки можуть змінюватися на місці через їхні методи `append()` та `extend()`, а також шляхом присвоєння через зрізи та індексацію.

Найкраще уявляти собі словники як невпорядкований набір пар *ключ:значення*, де ключі не повинні повторюватися (в межах одного словника). Пара фігурних дужок створює пустий словник: `{}`. Додання в середині дужок списку розділених комами пар ключ:значення задає початкові пари словника. У такій формі словники також подаються на вивід.

Основні операції зі словником — збереження значення за певним ключем і отримання значення для даного ключа. Видалення пари ключ:значення можливе за допомогою оператора `del`. Якщо ви зберігаєте значення для ключа, що вже існує, то старе значення, прив'язане до ключа "забувається". Спроба отримання значення для неіснуючого ключа призводить до помилки.

Метод об'єкта словника `keys()` повертає список всіх ключів, що використовуються у словнику, у випадковому порядку (якщо їх потрібно впорядкувати, то слід просто застосувати метод `sort()` для списку ключів). Щоб перевірити, чи присутній

певний ключ у словнику, слід використовувати метод словника `has_key()`[1].

Ось невеличкий приклад використання словника:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
True
```

Конструктор `dict()` створює словники зі списків пар ключ — значення, що задані кортежами. Якщо пари утворюються за певним повторюваним принципом, список ключів та значень може задаватися включеними списками:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack',
4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in vec])
{2: 4, 4: 16, 6: 36}
```

Далі у цьому підручнику ми познайомимося з конструкцією генератор, яка навіть краще підходить для задання пар ключ — значення для конструктора `dict()`.

Способи перебору

При переборі значень словників, ключі та відповідні значення можуть отримуватися за допомогою методу `iteritems()`:

```
>>> knights = {'Ґаллагад': 'Чистий', 'Робін':
'Хоробрий'}
>>> for k, v in knights.iteritems():
...     print k, v
...
Ґаллагад Чистий
Робін Хоробрий
```

При переборі послідовності, індекс та відповідне значення можуть бути отримані одночасно за допомогою функції `enumerate()`:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

При переборі двох чи більше послідовностей одночасно, їхні елементи можуть спаровуватися за допомогою функції `zip()`.

```
>>> questions = ["ім'я", 'пошук', 'колір']
>>> answers = ['Ланселот', 'святий Ґрааль',
'блакитний']
>>> for q, a in zip(questions, answers):
...     print '%s? %s' % (q, a)
...
ім'я? Ланселот
пошук? святий Ґрааль
колір? блакитний
```

Щоб перебрати послідовність в зворотньому порядку, спочатку треба вказати цю послідовність в звичайному порядку, а потім викликати функцію `reversed()`.

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
```

```
5
3
1
```

Щоб перебрати послідовність у певному порядку, слід використовувати функцію `sorted()`, що повертає новий впорядкований список, залишаючи вихідний список незміненим.

```
>>> basket = ['apple', 'orange', 'apple', 'pear',
'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
...
apple
banana
orange
pear
```

Докладніше про умови

Умови, що використовуються у твердженнях `while` та `if` можуть містити в собі будь-які оператори, а не лише порівняння.

Оператори порівняння `in` та `not in` перевіряють, чи присутній (або чи відсутній) певний елемент у послідовності.
Оператори `is` та `is not` порівнюють, чи два об'єкти є насправді тим самим об'єктом; це має значення лише для змінних об'єктів, скажімо, списків. Усі оператори порівняння мають однаковий пріоритет операцій, який є нижчим ніж у всіх числових операторів.

Порівняння можуть накопичуватися. Наприклад, `a < b == c` перевіряє, чи `a` менше за `b` і при цьому чи `b` дорівнює `c`.

Порівняння можуть комбінуватися з Булевими операторами `and` та `or`, а результат порівняння (чи будь-який

інший Булевий вираз) може заперечуватися через `not`. Вони мають менший пріоритет операцій ніж оператори порівняння. Серед них `not` має найвищий пріоритет, а `or` — найнижчий. Таким чином, `A and not B or C` еквівалентно `(A and (not B)) or C`. Звичайно, дужки можуть використовуватися для опису потрібного групування.

Булеві оператори `and` та `or` — це так звані оператори *короткого замкнення* (*short-circuit* operators): їхні аргументи обчислюються зліва направо і обчислення зупиняється як тільки стає відомим результат. Наприклад, якщо `A` і `C` істині, але `B` — ні, то `A and B and C` не обчислює вираз `C`. Загалом, значення, повернуте оператором короткого замкнення (коли воно використовується як загальне, а не як булеве значення) є останнім обчисленим аргументом.

Результат порівняння чи іншого Булевого виразу може бути присвоєно змінній. Наприклад:

```
>>> string1, string2, string3 = '', 'Trondheim',
'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Зауважте, що в Пайтоні, на відміну від C, присвоєння не може зустрічатися всередині виразів. Програмісти мови C можуть бути цим не задоволені, але це допомагає запобігти цілому класу проблем, що зустрічаються в програмах, написаних на C, коли у виразі набрано `=` замість `==`.

Порівняння послідовностей та інших типів

Об'єкти послідовностей можуть порівнюватися з іншими об'єктами, що належать до того самого типу послідовності. Порівняння робиться шляхом *лексикографічного* впорядкування: спочатку порівнюються перші два елементи, і якщо вони різняться, то це і визначає результат порівняння, якщо ж ні — то порівнюються наступні два елементи, і так далі аж до кінця послідовності. Якщо два елементи, що порівнюються, є в свою чергу послідовностями

одного типу, то лексикографічне порівняння продовжується рекурсивно. Якщо всі елементи послідовностей однакові, то і самі послідовності вважаються однаковими. Якщо одна з послідовностей є початком іншої, то коротша послідовність і є меншою. Лексикографічне впорядкування відбувається на основі впорядкування кодування ASCII для окремих символів. Ось окремі приклади порівняння однакових за типом послідовностей:

```
(1, 2, 3) &lt; (1, 2, 4)
[1, 2, 3] &lt; [1, 2, 4]
'ABC' &lt; 'C' &lt; 'Pascal' &lt; 'Python'
(1, 2, 3, 4) &lt; (1, 2, 4)
(1, 2) &lt; (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) &lt; (1, 2, ('abc', 'a'), 4)
```

Зауважте, що порівняння різнотипних об'єктів дозволяється. Результат детермінований, але довільний: типи впорядковуються за назвою. Таким чином, список (list) завжди менший за рядок (string), що завжди менший за кортеж (tuple) тощо. Змішані числові величини порівнюються відповідно до їхнього значення, отже 0 дорівнює 0.0 і т.д. (Зауваження: не варто покладатися на правила порівняння різнотипних об'єктів, тому що ці правила можуть змінитися в наступних версіях мови).

Загальні зауваження

Якщо ви залишите інтерпретатор Пайтона, а потім увійдете в нього знову, усі зроблені вами визначення функцій та змінних буде втрачено. Тому, якщо ви хочете написати дещо довшу програму, то для цього краще використовувати текстовий редактор і виконати програму, записану у файлі. Це зветься створенням *скрипта*. Коли програма збільшується, то ви можливо захочете розбити її на кілька файлів, щоб полегшити її підтримку. Можливо, ви маєте кілька зручних функцій, що ви створили у різних програмах, і ви хочете їх використати без копіювання їхніх описів у різні програми.

Для підтримки цього, Пайтон має певний механізм для створення визначень у файлі, які згодом можуть використовуватися у скрипті чи у діалоговому режимі інтерпретатора. Такий файл зветься *модулем*. Визначення, задані в модулі, *імпортуються* в інші модулі або

в *основний* (*main*) модуль, який являє собою сукупність об'єктів, до яких ви маєте доступ у скрипті, що виконується на найвищому рівні в режимі калькулятора.

Модуль — це файл, що складається з описів функцій та інструкцій Пайтона. Назва файла є назвою модуля, до якої додається розширення `.py`. Всередині модуля його назва доступна доступна через значення глобальної змінної `__name__`. Для прикладу, створіть у своєму улюбленому текстовому редакторі файл `fibo.py` у поточній директорії з таким змістом:

```
# Модуль, що обчислює числа Фібоначчі
def fib(n): # виводить числа Фібоначчі до n
    a, b = 0, 1
    while b &lt; n:
        print b,
        a, b = b, a+b
def fib2(n): # повертає числа Фібоначчі до n
    result = []
    a, b = 0, 1
    while b &lt; n:
        result.append(b)
        a, b = b, a+b
    return result
```

Тепер відкрийте інтерпретатор Пайтона та імпортуйте цей модуль за допомогою такої команди:

```
>>> import fibo
```

Ця команда додає в поточний простір імен лише саму назву модуля `fibo`, а не назви функцій, визначених у ньому. Використовуючи назву модуля ви можете дістатися до його функцій:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Якщо ви часто використовуватимете функцію, то їй можна дати локальну назву:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Докладніше про модулі

Модуль може містити не лише описи функцій, а й виконувані інструкції. Ці інструкції потрібні для ініціалізації модуля. Вони виконуються лише при *першому* імпортуванні модуля. (Насправді визначення функцій — це також інструкції, що "виконуються"; виконання ж полягає в тому, що назва функції вводиться у глобальний простір імен модуля).

Кожен модуль має свій власний простір імен, який використовується як глобальний усіма визначеними у цьому модулі функціями. Таким чином, автор модуля може використовувати глобальні змінні всередині модуля, не боячись можливого конфлікту з глобальними змінними, що задані користувачем модуля. З іншого боку, якщо ви знаєте, що робите, ви можете дістатися до глобальних змінних модуля за допомогою тієї ж нотації, що використовується для доступу функцій: `назва_модуля.елемент_модуля`.

Модулі можуть імпортувати інші модулі. Зазвичай, хоча це і не є необхідним, всі інструкції `import` пишуть на початку модуля (чи скрипта, якщо вам така назва більше подобається). Імпортовані назви модулів додаються до глобального простору імен модуля.

Існує варіант інструкції `import`, що напряму імпортує назви з модуля у простір імен імпортуючого модуля. Наприклад:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Ця операція не додає до локальної символьної таблиці назву самого модуля, з якого відбувся імпорт (зокрема, у цьому прикладі назва `fibo` — невизначена).

Існує також можливість для імпорту всіх назв, визначених у модулі:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

При цьому імпортуються усі назви, крім тих, що починаються з символу підкреслювання (`_`).

Шлях пошуку модулів

Якщо імпортується модуль, що зветься `spam`, то інтерпретатор спочатку шукає файл з назвою `spam.py` у поточній директорії, а потім у директоріях, визначених змінною середовища `PYTHONPATH`. Вона має такий же синтаксис, як і змінна оболонки `PATH`, тобто являє собою список директорій. Якщо `PYTHONPATH` не задано, або якщо файл там не знайдено, то пошук продувжується за типовою адресою, яка залежить від інсталяції; у системах Unix це здебільшого `/usr/local/lib/python`.

Тут слід уточнити, що пошук модулів починається зі списку директорій, заданих змінною `sys.path`, яка ініціалізується директорією, де розміщено скрипт вводу (чи у поточній директорії), а потім доповнюється з `PYTHONPATH` та залежним від інсталяції шляхом. Це дозволяє програмам змінювати адресу пошуку. Зауважте, що оскільки назва директорії, де знаходиться виконуваний скрипт, є у змінній `sys.path`, то важливо, щоб назва скрипта не збігалася з назвою певного стандартного модуля, бо інакше Пайтон спробує завантажити скрипт замість модуля при імпорті. Загалом це повинно призвести до помилки. Див. розділ "Стандартні модулі" для докладнішої інформації.

"Компільовані" файли

Існує можливість значного прискорення запуску коротких програм, що використовують багато стандартних модулів: якщо в директорії, де знаходиться файл `spam.py`, існує файл `spam.pyc`, то вважається, що він містить скомпільовану в байт-код версію модуля `spam`. Час останьої зміни версії `spam.py`, що використовується для створення `spam.pyc`, записується у `spam.pyc`, і файл `.pyc` пропускається, якщо час зміни скомпільованої версії не відповідає текстовій.

У більшості випадків для створення файла `spam.pyc` взагалі не потрібно нічого робити. Як тільки `spam.py` скомпільовано без проблем, інтерпретатор зробить спробу записати скомпільовану версію у `spam.pyc`. Якщо ця спроба не вдається, то це не призводить до помилки. Якщо з певних причин файл не записано повністю, то новостворений `spam.pyc` буде вважатися недійсним і таким чином не буде використовуватися пізніше. Вміст файла `spam.pyc` не залежить від платформи, отже директорія, що містить модулі, написані на Пайтоні, може використовуватися машинами різних архітектур.

Окремі поради для експертів:

- Якщо інтерпретатор викликається з опцією **-O**, то оптимізований код буде створено і записано в файлах з розширенням `.pyo`. Наразі оптимізатор не дуже допомагає, він лише видаляє інструкції `assert`. При використанні **-O** оптимізуються *усі* байт-коди; файли `.pyc` пропускаються, а з файлів `.py` компілюється оптимізований байт-код.
- Подання подвійної опції **-O** інтерпретатору Пайтона (**-OO**) призведе до оптимізації, що в окремих випадках може спричинити неполадки у програмі. Наразі видаляються лише рядки `__doc__`, що призводить до створення більш компактних файлів `.pyo`. Оскільки програми можуть покладатися на існування цих змінних, цю опцію слід лише використовувати лише якщо ви точно знаєте, що робите.
- Швидкість виконання програми не змінюється, коли її зчитано з файлів `.pyc` чи `.pyo`, а не `.py`. Єдине, що відбувається швидше — це завантаження програми.
- Якщо скрипт запускається через виклик по імені з командного рядка, то його байт-код ніколи не записується у файл `.pyc` чи `.pyo`. Таким чином, прискорити запуск скрипта можна, якщо помістити більшість коду в окремий модуль і створити невеликий стартовий файл, що імпортує той модуль. Також можна ввести назву файла `.pyc` чи `.pyo` з командного рядка.
- Можна також викликати файл `spam.pyc` (чи `spam.pyo`) навіть якщо файл `spam.py` для цього ж модуля відсутній. Це може використовуватися для розповсюдження бібліотеки коду Пайтона, з якого не зовсім просто відтворити код-джерело.

- Модуль `compileall` може створити файли `.pyc` (чи `.pyo`, якщо використано **-O**) для всіх модулів, що знаходяться у певній директорії.

Стандартні модулі

Пайтон має бібліотеку стандартних модулів, яка описана в окремому документі, що зветься "Довідник бібліотеки мови Python" (Python Library Reference) (надалі "Довідник бібліотеки"). Окремі модулі, що вбудовано в інтерпретатор, надають доступ до операцій, що не є основною частиною мови, але все таки вони вбудовані, чи то задля ефективності, чи для того, щоб надати доступ до функцій операційної системи, зокрема системних викликів. Такі модулі є частиною конфігурації, яка залежить від платформи. Наприклад, модуль `amoeba` існує лише у системах, що певним чином підтримують примітиви системи Amoeba. Один окремий модуль заслуговує спеціальної уваги: `sys`, що вбудований у будь-який інтерпретатор мови Пайтон. Змінні `sys.ps1` та `sys.ps2` визначають рядки, що використовуються як головне та другорядне запрошення:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C&gt; '
C&gt; print 'Yuck!'
Yuck!
C&gt;
```

Ці дві змінні визначені лише коли інтерпретатор працює в діалоговому режимі.

Змінна `sys.path` — це список рядків, що визначають використовувані інтерпретатором шляхи пошуку модулів. Вона ініціалізується стандартним значенням, що береться зі змінної середовища `PYTHONPATH` або із вбудованого стандартного значення (якщо `PYTHONPATH` не задано). ЇЇ можна змінити за допомогою стандартних спискових операцій:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

Функція `dir()`

Вбудована функція `dir()` використовується для виявлення усіх назв, визначених у модулі. Вона повертає впорядкований список рядків:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__',
'__name__', '__stderr__',
 '__stdin__', '__stdout__', '_getframe', 'api_version',
'argv',
 'builtin_module_names', 'byteorder', 'callstats',
'copyright',
 'displayhook', 'exc_clear', 'exc_info', 'exc_type',
'excepthook',
 'exec_prefix', 'executable', 'exit',
'getdefaultencoding', 'getdlopenflags',
 'getrecursionlimit', 'getrefcount', 'hexversion',
'maxint', 'maxunicode',
 'meta_path', 'modules', 'path', 'path_hooks',
'path_importer_cache',
 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval',
'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr',
'stdin', 'stdout',
 'version', 'version_info', 'warnoptions']
```

Без аргументів `dir()` повертає назви, визначені на момент виклику функції:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

Зауважте, що цей список містить всі види ідентифікаторів: змінні, модулі, функції тощо.

`dir()` не видає назв вбудованих функцій та змінних. Якщо ці назви потрібні, то вони визначені у стандартному модулі `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
 'DeprecationWarning', 'EOFError', 'Ellipsis',
'EnvironmentError',
 'Exception', 'False', 'FloatingPointError', 'IOError',
'ImportError',
 'IndentationError', 'IndexError', 'KeyError',
'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None',
'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
'OverflowWarning',
 'PendingDeprecationWarning', 'ReferenceError',
 'RuntimeError', 'RuntimeWarning', 'StandardError',
'StopIteration',
 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError',
 'True', 'TypeError', 'UnboundLocalError',
'UnicodeError', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError',
'__debug__', '__doc__',
 '__import__', '__name__', 'abs', 'apply', 'bool',
'buffer',
 'callable', 'chr', 'classmethod', 'cmp', 'coerce',
'compile', 'complex',
 'copyright', 'credits', 'delattr', 'dict', 'dir',
'divmod',
 'enumerate', 'eval', 'execfile', 'exit', 'file',
'filter', 'float',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex',
'id',
 'input', 'int', 'intern', 'isinstance', 'issubclass',
'iter',
 'len', 'license', 'list', 'locals', 'long', 'map',
'max', 'min',
 'object', 'oct', 'open', 'ord', 'pow', 'property',
'quit',
```

```
 'range', 'raw_input', 'reduce', 'reload', 'repr',
'round',
 'setattr', 'slice', 'staticmethod', 'str', 'string',
'sum', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange',
'zip']
```

Пакети

Пакети — це засіб організації іменного простору модулів Пайтона шляхом використання "нотації крапками" ("dotted module names). *Наприклад, назва модуля `A.B` вказує на підмодуль "`B`" у пакеті "`A`". Подібно до того як використання модулів позбавляє авторів зайвих тривог щодо назв глобальних змінних, визначених кимось іншим, використання назв модулів, з'єднаних крапками, допомагає авторам багатомодульних пакетів (як, скажімо, NumPy чи графічної бібліотеки Python Imaging Library) уникати проблем, пов'язаних з використанням однакових назв модулів.*

Скажімо, ви хочете створити колекцію модулів (пакет) для обробки звукових даних та звукових файлів. Існує багато різних звукових форматів файлів (здебільшого вони розпізнаються за розширенням, наприклад: `.wav`, `.aiff`, `.au`) і вам потрібно створити і утримувати зростаючу колекцію модулів для конвертування різних звукових форматів. Існують також численні операції для обробки звукових даних (наприклад, накладання реверберації, застосування еквалайзера, створення штучного стереоефекту, тощо), отож крім усього іншого вам доведеться створювати нескінченний потік модулів для здійснення усіх цих операцій. Тут представлено можливу структуру вашого пакета (виражену у термінах ієрархічної файлової системи):

```
Sound/ Пакет найвищого рівня
      __init__.py Ініціалізація звукового пакета
      Formats/ Підпакет конвертування форматів файлів
          __init__.py
          wavread.py
          wavwrite.py
          aiffread.py
          aiffwrite.py
          auread.py
```

```
        auwrite.py
        ...
    Effects/ Підпакет звукових ефектів
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
    Filters/ Підпакет фільтрів
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
        ...
```

При імпортуванні пакета Пайтон шукає його піддиректорію, перебираючи директорії, що містяться у `sys.path`.

Файли `__init__.py` потрібні, щоб Пайтон міг розпізнати директорії, як такі, що містять в собі пакети; це зроблено з метою запобігання випадковому заміщенню інших модулів, розташованих глибше в дереві каталогів, директоріями, що мають загальні назви (наприклад, "`string`"). У найпростішому випадку `__init__.py` може бути порожнім файлом, але він може виконати ініціалізацію пакета чи задати змінну `__all__`, описану далі.

Користувачі пакета можуть імпортувати окремі модулі з пакета, наприклад:

```
import Sound.Effects.echo
```

Ця команда завантажує підмодуль `Sound.Effects.echo`. Посилання на нього повинно робитися через його повну назву.

```
Sound.Effects.echo.echofilter(input, output, delay=0.7,
atten=4)
```

Альтернативний спосіб імпортування підмодуля такий:

```
from Sound.Effects import echo
```

Це також завантажує підмодуль `echo`, а також уможливлює посилання на нього без префіксу пакета, тобто його можна використовувати таким чином:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Можливо також імпортувати бажану функцію чи змінну напряму:

```
from Sound.Effects.echo import echofilter
```

При цьому також завантажується підмодуль `echo`, але в той же час функція `echofilter()` може використовуватися напряму:

```
echofilter(input, output, delay=0.7, atten=4)
```

Зауважте, що при використанні `from пакет import елемент`, елемент може бути підмодулем (чи підпакетом) пакета або будь-якою іншою назвою, визначеною в пакеті, як, скажімо, функція, клас чи змінна. Інструкція `import` спершу перевіряє, чи визначено заданий елемент у пакеті; якщо ні — цей елемент вважається модулем, і робиться спроба його завантаження. Якщо ж і модуля не знайдено,то генерується "помилка імортування" `ImportError`.

З іншого боку, при використанні синтаксису `import елемент.піделемент.підпіделемент`, кожен елемент окрім останього повинен бути пакетом. Останній елемент може бути модулем чи пакетом, але не може бути класом, функцією чи змінною, визначеною у попередньому елементі.

Імпортування * з пакета

Що ж відбувається, коли користувач пише `from Sound.Effects import *`? В ідеальному випадку можна сподіватися, що тут певним чином відбувається пошук по фаловій системі, знайдено всі підмодулі пакета, і всіх їх імпортовано. На жаль, ця операція не дуже добре працює на платформах Mac та Windows, де файлова система не розрізняє великі та малі літери в назві файлу. На цих платформах немає певного способу визначення, яким саме чином повинен бути імпортований файл `ECHO.PY`: як модуль `echo`, `Echo` чи `ECHO`.

Єдиним вирішенням проблеми є явне задання автором змісту пакета. Інструкція import використовує таку конвенцію: якщо код пакета `__init__.py` визначає список, що називається `__all__`, то він вважається списком назв модулів, що потрібно імпортувати при використанні інструкції `from пакет import *`. Створення повного списку модулів залишається на відповідальність автора. Автори пакетів можуть також обмежити `import *` зі своїх пакетів. Наприклад, файл `Sounds/Effects/__init__.py` може містити такий код:

```
__all__ = ["echo", "surround", "reverse"]
```

Це означає, що `from Sound.Effects import *` імпортує лише три вказані підмодулі пакета `Sound`.

Якщо `__all__` не визначено, то інструкція `from Sound.Effects import *` *не* імпортує усі підмодулі пакета `Sound.Effects` у поточний простір імен, а лише забезпечує імпорт пакета `Sound.Effects` (можливо шляхом його ініціалізації з `__init__.py`) і потім імпортує всі назви, визначені в пакеті. Це включає всі назви визначені (та явно завантажені) у файлі `__init__.py`. Це також включає будь-які підмодулі пакета, що були явно завантажені попередніми інструкціями імпортування. Розгляньмо такий код:

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

У цьому прикладі модулі echo та surround імпортуються у поточний простір імен тому, що вони визначені у пакеті `Sound.Effects` при виконанні інструкції `from ... import`. (Це діє також, коли визначено `__all__`).

Зауважте, що звичка імпортувати з модулів все за допомогою символа "`*`" не заохочується, оскільки це часто призводить до нечитабельного коду. Хоча це можна використовувати, щоб зберегти час, потрібний для вводу під час діалогового режиму, до того ж певні модулі навмисно створені, щоб експортувати модулі відповідно до певних зразків.

Пам'ятайте, що немає нічого поганого, якщо ви використовуєте `from Пакет import певний_підмодуль`! Загалом, саме така форма імпортування є рекомендованою, за винятком коли імпортуючий модуль потребує підмодуль з тією самою назвою, але з іншого пакета.

Внутрішньопакетні посилання

Підмодулі часто посилаються один на одного. Наприклад, модуль `surround` може посилатися на модуль `echo`. Такі посилання є настільки поширеними, що інструкція `import` спочатку перевіряє зовнішній[1] пакет перед тим, як розпочати пошук в `sys.path`. Таким чином, модуль `surround` може просто використовувати `import echo` чи `from echo import echofilter`. Якщо імпортований модуль не знайдено в поточному пакеті (для якого поточний модуль є підмодулем), то інструкція `import` шукає модуль найвищого рівня[2] з цією назвою.

Якщо пакети зорганізовано у підпакети (як у наведеному прикладі з пакетом `Sound`), то посилання «навпростець» на пакет-близнюк неможливе — тут повинна використовуватися повна назва пакета. Наприклад, якщо модулю `Sound.Filters.vocoder` потрібен модуль `echo` у пакеті `Sound.Effects`, то для цього можна використовувати `from Sound.Effects import echo`.

Пакети, що знаходяться у різних директоріях

Пакети підтримують ще однин спеціальний атрибут: `__path__`. Він ініціалізується списком, який містить назву директорії, у якій знаходиться `__init__.py` даного пакета, перед виконанням коду, що знаходиться у цьому файлі. Ця змінна може бути змінена, що позначиться на майбутніх пошуках підпакетів даного пакета.

Хоча ця функція використовується нечасто, вона може використовуватися для розширення кількості модулів, що складають даний пакет.

Можливості форматування виводу

Раніше ми познайомилися з двома способами виводу значень: *твердження з виразами* та оператор `print`. (Третій спосіб — це використання методу `write()` файлових об'єктів; на файл стандартного виводу можна посилатися через `sys.stdout`. Див. "Опис бібліотеки" для подальшої інформації з цього приводу).

Часто потрібно більше контролю над форматуанням виводу, аніж простий друк списку значень розділених комами. Існують два способи форматування виводу. Перший — це самому створити рядки для виводу: за допомогою операцій зрізу та з'єднання можна отримати будь-яке можливе форматування. Стандартний модуль `string` має деякі корисні операції для заповнення рядків до певної ширини колонок, про що йтиметься незабаром. Другий спосіб — використання оператора `%` з рядком в якості лівого параметра. Оператор `%` інтерпретує лівий операнд подібно до функції `sprintf()` і застосовує до нього правий аргумент, повертаючи при цьому рядок, що є результатом операції форматування.

Тоді лишається таке питання: яким чином можна конвертувати різні значення у рядки? На щастя, Пайтон має функції для конвертування будь-яких значень у рядки: `repr()` та `str()`. Зворотні лапки (``` `` ```) еквівалентні функції `repr()`, але їхне вживання небажане.

Функція `str()` створює текст який зручно читати людині, тоді як `repr()` — текст призначений для інтерпритатора. Для об'єктів, що не мають читабельного представлення, `str()` поверне такий самий текст, як і `repr()`. Багато значень, скажімо, числа чи такі структури як списки чи словники, мають однакову репрезентацію при застосуванні будь-якої з цих функцій. Рядки та числа з рухомою крапкою мають дві різні репрезентації.

Окремі приклади:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> s1='привіт, світе\n' # кодування UTF-8
```

```
>>> str(s1)
'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd1\x96\xd1\x82,
\xd1\x81\xd0\xb2\xd1\x96\xd1\x82\xd0\xb5\n'
>>> repr(s1)
"'\\xd0\\xbf\\xd1\\x80\\xd0\\xb8\\xd0\\xb2\\xd1\\x96\\xd1
\\x82,
\\xd1\\x81\\xd0\\xb2\\xd1\\x96\\xd1\\x82\\xd0\\xb5\\n'"
>>> print s1
привіт, світе

>>> str(0.1)
'0.1'
>>> repr(0.1)
'0.10000000000000001'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'x дорівнює ' + repr(x) + ', а y — ' + repr(y) +
'...'
>>> print s
x дорівнює 32.5, а y — 40000...
>>> # Функція repr() додає лапки та зворотні косі:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # Аргументом функції repr() може бути будь-який
об'єкт:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
>>> # зворотні лапки зручні у діалоговому режимі:
... `x, y, ('spam', 'eggs')`
"(32.5, 40000, ('spam', 'eggs'))"
```

Ось два способи виводу таблиці квадратів та кубів:

```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Зауважте кому в кінці попереднього рядка
...     print repr(x*x*x).rjust(4)
...
 1  1  1
 2  4  8
```

```
 3  9  27
 4  16  64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10  100  1000
>>> for x in range(1,11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
 1  1  1
 2  4  8
 3  9  27
 4  16  64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10  100  1000
```

(Зауважте, що один пробіл між колонками було додано функцією `print`, яка завжди друкує пробіли між аргументами).

Цей приклад демонструє метод `rjust()` рядкових об'єктів, який вирівнює рядки по правій стороні шляхом додання пробілів з лівої сторони. Існують також подібні методи для вирівнювання по центру (`center()`) та по лівій стороні (`ljust()`). Ці методи не здіснюють виводу, а лише повертають новий рядок. Якщо рядок, що вирівнюється, задовгий — то вони повертають його як є, без скорочення. При цьому вигляд колонок буде спотворено, але здебільшого це краще, ніж вивід хибного значення. (Якщо скорочення значення все-таки потрібне, то цього можна завжди досягти за допомогою операції зрізу: "`x.ljust(n)[:n]`").

Існує також інший метод, `zfill()`, що додає зліва до рядка, що виражає число нулі. Цей метод також розуміє знаки плюс та мінус:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
```

```
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Використання оператора `%` виглядає таким чином:

```
>>> import math
>>> print 'Пі приблизно дорівнює %5.3f.' % math.pi
Пі приблизно дорівнює 3.142.
```

Якщо треба сформатувати в рядок більш ніж одне значення, то правий операнд повинен задаватися кортежем, як у цьому прикладі:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '%-10s ==&gt; %10d' % (name, phone)
...
Jack ==&gt;       4098
Dcab ==&gt;       7678
Sjoerd ==&gt;     4127
```

Більшість форматувань працюють точнісінько як і в C і потребують належного типу, але при передачі неправильного типу ви отримуєте виняток, а не фатальну помилку (core dump). Формат `%s` є найбільш гнучким: якщо відповідний аргумент не є рядком, то відбувається авоматична конверсія за допомогою вбудованої функції `str()`. Використання зірочки (`*`) для передачі ширини чи точності як окремого (цілочислового) аргумента також можливе. Ключі форматування мови C `%n` та `%p` не підтримуються.

Якщо потрібно сформатувати доволі довгий рядок, який ви не хочете розбивати на окремі підрядки, то набагато простіше посилатися на на параметри за допомогою назв, а не за їх порядком. Це можна зробити за допомогою нотації `%(назва)формат`, як показано нижче:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab':
8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab:
%(Dcab)d' % table
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Це особливо корисно у комбінації з новою вбудованою функцією `vars()`, яка повертає словник, що складається з усіх локальних змінних.

Зчитування і запис файлів

Функція `open()` повертає файловий об'єкт і здебільшого вживається з двома аргументами: "`open(назва_файлу, режим)`".

```
>>> f=open('/tmp/workfile', 'w')
>>> print f
&lt;open file '/tmp/workfile', mode 'w' at 80a0960&gt;
```

Перший аргумент — рядок, що є назвою файлу. Другий аргумент - також рядок, що містить кілька символів, котрі описують, як буде використовуватись файл. Режим може бути `'r'` (read), коли файл відкривається лише для зчитування, `'w'` (write) — для запису (існуючий файл з цією назвою буде зтерто), `'a'` (append) — для додання (усі нові дані будуть автоматично додані в кінець файла), а також `'r+'`, що відкриває файл для зчитування та запису одночасно. Аргумент, що задає режим не є обов'язковим, якщо його не вказано, то файл буде відкрито для зчитування.

На системах Windows та Macintosh `'b'`, доданий до режиму, вказує на те, що файл повинен бути відкрито у бінарному режимі. Таким чином на цих системах маємо такі додаткові режими: `'rb'`, `'wb'`, and `'r+b'`. Взаємодія Windows з текстовими та бінарними файлами відрізняється; символ нового рядка автоматично опускається при зчитуванні чи записі даних. Така позакулісна зміна файлових даних годиться для роботи з файлами у кодуванні ASCII, але вона зіпсує бінарні дані таких форматів як JPEG чи EXE. При зчитуванні чи записі таких файлів необхідно використовувати бінарний режим. (Зауважте, що точна семантика текстового режиму на Macintosh залежить від використовуваної бібліотеки мови C).

Методи файлових об'єктів

У цьому розділі припускається, що файловий об'єкт з назвою `f` вже було створено.

Зчитування змісту файла здійснюється за допомогою виклику `f.read(`*`розмір`*`)`, який зчитує певну кількість даних і повертає їх у формі рядка. *розмір* - це необов'язковий числовий аргумент. Якщо *розмір* пропущено або його значення від'ємне, то буде зчитано і повернуто увесь файл; якщо ж файл удвічі більний за пам'ять вашого комп'ютера — то це ваша особиста проблема. Якщо ж розмір вказано, то буде зчитано кількість байтів, що дорівнює або є меншою за вказану кількість. Якщо дойдено до кінця файла, `f.read()` поверне пустий рядок (`""`).

```
>>> f.read()
'Це увесь файл.\n'
>>> f.read()
''
```

`f.readline()` зчитує окремий рядок із файла; символ нового рядка (`\n`) залишається в кінці рядка (він може бути відсутнім лише в отаньому рядку, якщо файл не закінчується цим символом). Завдяки цьому повернене значення є однозначним; якщо `f.readline()` повертає пустий рядок, то це означає, що досягнуто кінця файла, тоді ж як пустий рядок репрезентовано символом `'\n'`, тобто рядком, що складається з символа переходу на новий рядок.

```
>>> f.readline()
'Це перший рядок файла.\n'
>>> f.readline()
'Другий рядок файла.\n'
>>> f.readline()
''
```

`f.readlines()` повертає список, що складається з усіх рядків файла. Якщо цей метод отримує необов'язковий параметр *бажаний_розмір*, то буде зчитано вказану кількість байтів, плюс стільки, скільки потрібно, щоб завершити рядок. Це часто використовується для швидкого зчитування великих файлів радок за рядком без завантаження цілого файла в пам'ять комп'ютера. Повертаються лише цілі рядки.

```
>>> f.readlines()
['Це перший рядок файла.\n', 'Другий рядок файла\n']
```

`f.write(`*`рядок`*`)` записує вміст *рядка* в файл, повертаючи `None`.

```
>>> f.write('Це — тест\n')
```

`f.tell()` повертає ціле число, що вказує поточну позицію у файлі, виміряну в байтах, починаючи від початку файла. Для зміни позиції файлового об'єкта слід використовувати "`f.seek(`*`скільки, звідки`*`)`". Позиція обчислюється шляхом додання значення, вираженого параметром *скільки*, до пункту відліку, що визначається параметром *звідки* і може мати такі значення: 0 (початок файла), 1 (поточна позиція), 2 (кінець файла). Параметр *звідки* може бути пропущено, при цьому пошук позиції відбудеться відносно початку файла.

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)  # Йдемо до 6-го байта у файлі
>>> f.read(1)
'5'
>>> f.seek(-3, 2)  # Йдемо до 3-го байта з кінця
>>> f.read(1)
'd'
```

Після того, як роботу з файлом закінчено, слід викликати `f.close()` для його закриття і звільнення системних ресурсів, що використовуються відкритим файлом. Після виклику `f.close()` спроба використання файлового об'єкта призведе до помилки.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
 File "&lt;stdin&gt;", line 1, in ?
ValueError: I/O operation on closed file
```

Файлові об'єкти мають такі додаткові методи, як `isatty()` та `truncate()`, що використовуються набагато рідше. Див. "Опис бібліотеки" для докладнішої інформації, що стосується файлових об'єктів.

Модуль `pickle`

Текст може легко записуватися у файл та зчитуватися з нього. Певного зусилля потребують числа, тому що `read()` повертає лише текст, який треба обробити за допомогою певної функції, скажімо, `int()`, котра отримує рядок (напр, "123") і повертає його числове значення (123). При записі складніших типів даних (списків, словників чи реалізацій класів) ситуація значно ускладнюється.

Щоб не завдавати користувачам зайвих турбот, пов'язаних зі створенням та налаштуванням коду для збереження складних типів даних, Пайтон має стандартний модуль, що зветься `pickle`. Це чудовий модуль, що може взяти будь-який об'єкт мови Пайтон (і навіть певні види коду!), і створити його рядкове представлення. Цей процес зветься *запаковуванням* (*pickling*, дослівно "*маринування*"). Реконструкція об'єкта з його рядкового представлення зветься *розпаковуванням* (*unpickling*). Між цими двома операціями рядкове представлення об'єкта може бути збережене у файлі чи переслане через мережу до віддаленої машини.

Якщо існує об'єкт `x`, і файловий об'єкт `f`, відкритий для запису, то найпростіший спосіб запаковування об'єкта потребує одного-єдиного рядка коду:

```
pickle.dump(x, f)
```

Для зворотньої дії, якщо `f` — файловий об'єкт, відкритий для зчитування:

```
x = pickle.load(f)
```

(Існують також можливості для запаковування кількох об'єктів одночасно, для збереження результату не в файлі тощо. Докладнішу інформацію можна отримати у документації модуля `pickle`)

`pickle` — це стандартний спосіб зберігання коду та його подальшого використання іншими програмами чи іншим викликом цієї ж програми. Технічний термін для цього явища — персистентний об'єкт. Через те що `pickle` дуже широко використовується, численні автори, що

створюють розширення для Пайтона, перевіряють чи нові типи даних (наприклад, матриці) можуть нормально запаковуватися та розпаковуватися.

Помилки та винятки

Раніше про помилки згадувалося лише побіжно, але якщо ви пробували виконувати приклади, подані раніше, то напевно вже бачили деякі з них. Існують два окремі типи помилок: *синтаксичні помилки* та *винятки*.

Синтаксичні помилки

Синтаксичні помилки є можливо найбільшою проблемою коли ви починаєте вивчати мову Пайтон:

```
>>> while True print 'Привіт, світе'
  File "stdin", line 1, in ?
  while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

Парсер повторює рядок, де було порушено правило, і виводить маленьку "стрілку", яка вказує на місце, де помилку було вперше помічено. Ця помилка була спричинена знаком (або принаймні помічена там), що передує стрілці : у цьому прикладі помилку знайдено перед ключовим словом `print`, тому що перед ним було пропущено двокрапку (":").

Також виводиться назва файла та номер рядка, щоб ви знали, де саме відбулася помилка, якщо ввід було отримано зі скрипта.

Винятки

Навіть коли інструкція чи вираз синтаксично правильні, їх виконання може призвести до помилки. Помилки, що відбуваються під час виконання програми звуться *винятками* і вони не є безумовно фатальними: незабаром ви довідаєтеся, як їх можна фільтрувати у програмах, написаних на Пайтоні. Якщо помилка не обробляється програмою, то тоді її результатом буде повідомлення на зразок тих, що показані нижче:

```
>>> 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
```

Останній рядок повідомлення про помилку вказує на те, що ж саме трапилося. Винятки належать до різних типів і назва типу виводиться як частина повідомлення. У наведених вище прикладах типами помилок є `ZeroDivisionError` (ділення на нуль), `NameError` (помилка імені) та `TypeError` (помилка типу). Назва помилки, що виводиться і є назвою вбудованого винятку, що відбувся. Це справедливо для всіх вбудованих винятків, але не завжди — для винятків, заданих користувачем (хоча це і корисна конвенція). Назви стандартних винятків — це вбудовані ідентифікатори (а не зарезервовані ключові слова).

Решта тексту в повідомленні — деталі, інтерпретація та значення яких залежить від типу помилки.

Фільтрування помилок

Можна писати програми, здатні обробляти задані помилки. Розгляньмо наступний приклад, де програма вимагає вводу допоки не буде введено ціле число, але при цьому дозволяє користувачеві перервати програму (використовуючи `Control-C` чи щось інше, що розуміє операційна система); зауважте, що переривання, викликане користувачем, супроводжується створенням винятку `KeyboardInterrupt`.

```
>>> while True:
...  try:
...   x = int(raw_input("Введіть, будь-ласка, ціле число:
"))
...   break
...  except ValueError:
...   print "Неправильне значення. Спробуйте знову..."
...
```

Інструкція `try` працює таким чином:

- Спочатку виконується блок, розташований між ключовими словами `try` та `except`.
- Якщо не трапилось жодного винятку, то інструкції, розташовані за `except` пропускаються і таким чином закінчується виконання блоку `try`.
- Якщо виняток трапився під час виконання однієї з інструкцій блоку `try`, то решта його інструкцій пропускається.

  Після цього, якщо тип помилки збігається з типом винятку, зазначеним після ключового слова `except`, то виконуються інструкції блоку `except`, після чого продовжується виконання коду, розташованого за оператором `try`.
- Якщо виняток, що відбувся, не відповідає винятку, вказаному після ключового слова `except`, то він передається

зовнішнім операторам `try`. Якщо виняток так і не було відфільтровано, то він вважається *неопрацьованим винятком* і виконання програми зупиняється із виведенням помилки, як показано вище.

Твердження `try` може мати більш ніж одну конструкцію `except` для обробки різних помилок, але не більше ніж одну з них буде виконано. Оброблено буде лише помилки, що відбулися у відповідному блоці `try`, а не всередпні самих фільтрів. Конструкція `except` може опрацьовувати кілька винятків одночасно; при цьому їх назви оточуються дужками:

```
... except (RuntimeError, TypeError, NameError):
...   pass
```

Остання конструкція `except` може не вказувати назви винятків, що може використовуватися для обробки будь-якої помилки. Цю функцію слід використовувати з надзвичайною обережністю, бо з її допомогою можна легко приховати справжні помилки у програмі! Вона також може служити для виводу повідомлення про помилку з повторним підняттям помилки (що може використовуватися для її подальшої обробки в середовищі, яке викликало даний код):

```
import sys
```

```
try:
 f = open('myfile.txt')
 s = f.readline()
 i = int(s.strip())
except IOError, (errno, strerror):
 print "Помилка вводу/виводу (%s): %s" % (errno,
strerror)
except ValueError:
 print "Неможливо конвертувати дані в ціле число."
except:
 print "Несподівана помилка:", sys.exc_info()[0]
 raise
```

>>>>>>>>> Конструкція `try` ... `except` має необов'язкову конструкцію *else ("інакше")*, яка, якщо присутня, повинна закривати всі конструкції `except`. Це корисно для створення коду, що має виконатися в разі, якщо не відбулося жодного винятку. Наприклад:

```
for arg in sys.argv[1:]:
 try:
  f = open(arg, 'r')
 except IOError:
  print 'неможливо відкрити', arg
 else:
  print arg, 'має', len(f.readlines()), 'рядків'
  f.close()
```

Вживання блоку `else` — краще, ніж створення додаткового коду всередині блоку `try`, тому що дозволяє уникати випадкової обробки винятку, який не повинен був підніматися всередині коду, захищеного твердженням `try` ... `except`.

Виняток може мати певну величину, що асоціюється з ним. Ця величина також відома як *аргумент* винятку. Присутність і тип цього аргументу залежать від типу винятку.

Блок `except` може визначати змінну (або список) після назви винятку. Ця змінна прив'язується до реалізації винятку, чиї аргументи зберігаються у `instance.args`. Для зручності, реалізація винятку визначає `__getitem__` та `__str__`, а отже вивід та доступ до аргументів може здійснюватися напряму без посилання на `.args`.

```
>>> try:
...  raise Exception('spam', 'eggs')
... except Exception, inst:
...  print type(inst) # реалізація винятку
...  print inst.args # аргументи збережено в .args
...  print inst # __str__ дозволяє виводити аргументи
напряму
...  x, y = inst # __getitem__ дозволяє розпаковувати
аргументи напряму
...  print 'x =', x
...  print 'y =', y
...
&lt;type 'instance'&gt;
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Якщо виняток має аргумент, то він виводиться як остання частина ("деталь") повідомлення для необроблених помилок.

Фільтри винятків обробляють не лише ті помилки, що відбулися безпосередньо у блоці `try`, але і ті, що виникають всередині функцій, які викликаються (навіть опосередковано) із блоку `try`. Наприклад:

```
>>> def this_fails():
...  x = 1/0
...
>>> try:
...  this_fails()
... except ZeroDivisionError, detail:
...  print 'Відбулася помилка при виконанні:', detail
...
Відбулася помилка при виконанні: integer division or
modulo
```

Створення винятків

Твердження `raise` дозволяє програмісту задавати винятки. Наприклад:

```
>>> raise NameError, 'ПривітВам'
```

```
Traceback (most recent call last):
 File "stdin;", line 1, in ?
NameError: ПривітВам
```

Перший аргумент для `raise` називає виняток, що створюється. Необов'язковий другий аргумент є аргументом самого винятку.

Якщо потрібно лише визначити, чи було піднято помилку, без подальшої її обробки, простіша форма твердження `raise` дозволяє заново підняти помилку:

```
>>> try:
...  raise NameError, 'ПривітВам'
... except NameError:
...  print 'Тут промайнув виняток!'
...  raise
...
Тут промайнув виняток!
Traceback (most recent call last):
 File "stdin", line 2, in ?
NameError: ПривітВам
```

Винятки, визначені користувачем

Програми можуть визначати нові винятки шляхом створення нового класу винятку. Загалом, винятки повинні походити із класу `Exception`, прямо чи опосередковано. Наприклад:

```
>>> class MyError(Exception):
...  def __init__(self, value):
...   self.value = value
...  def __str__(self):
...   return repr(self.value)
...
>>> try:
...  raise MyError(2*2)
... except MyError, e:
...  print 'Відбувся мій виняток,
{{error|{{error|велич}}}}ина:', e.value
...
Відбувся мій виняток, {{error|{{error|велич}}}}ина: 4
>>> raise MyError, 'oops!'
```

```
Traceback (most recent call last):
 File "stdin", line 1, in ?
__main__.MyError: 'oops!'
```

Класи винятків (як і звичайні класи) можуть виконуватимуть будь-що, але загалом вони створюються максимально простими і мають обмежену кількість атрибутів, для надання інформації про помилку, яка може використовуватися фільтрами винятків. При створенні модуля, що може відкинути кілька окремих помилок, традиційно створюється базовий клас для всіх винятків, визначених у цьому модулі, а також підклас для створення специфічних класів винятків для окремих помилок:

```
class Error(Exception):
 """Базовий клас для винятків цього модуля."""
 pass

class InputError(Error):
 """Винятки для помилок вводу.

 Атрибути:
 expression -- вираз вводу, де відбулася помилка
 message) -- повідомлення про помилку
 """

 def __init__(self, expression, message):
  self.expression = expression
  self.message = message

class TransitionError(Error):
 """Відкидається при недозволеній зміні стану.

 Атрибути:
 previous -- початковий стан переходу
 next -- пропонований наступний стан
 message -- пояснення, чому такий перехід не дозволяється
 """

 def __init__(self, previous, next, message):
  self.previous = previous
  self.next = next
  self.message = message
```

Більшість винятків мають назви, що закінчуються словом "Error", подібно до назв стандартних винятків.

Багато стандартних модулів визначають свої власні винятки для повідомлень про помилки, що можуть відбутися у визначених ними функціях. Докладніша інформація про класи подана в наступному розділі.

Визначення очищувальних дій

Твердження `try` ще один необов'язковий блок, який визначає різні "очищувальні" дії, що мають виконатися за будь-яких обставин. Наприклад:

```
>>> try:
...  raise KeyboardInterrupt
... finally:
...  print 'Прощай, світе!'
...
Прощай, світе!
Traceback (most recent call last):
 File "stdin", line 2, in ?
KeyboardInterrupt
```

Цей *заключний блок* виконується незалежно від того, чи відбувся виняток у блоці `try` . Якщо відбувся виняток, то він заново відкинеться після виконання блоку `finally`. Фінальний блок також виконується при виході з блоку `try` за допомогою тверджень `break` чи `return`.

Код у заключному блоці корисний для звільнення зовнішніх ресурсів (таких як файли чи зв'язки з мережею), незалежно від того, чи використання цих ресурсів було успішним, чи ні.

Твердження `try` повинно мати або один чи більше блоків `except`, або один блок `finally`, але не обидва разом.

Механізм створення класів у Пайтоні додає класи до мови з мінімальним використанням нового синтаксису та семантики. Він є сумішшю рис, що існують у мовах C++ та Modula-3. Як і модулі, класи у Пайтоні не створюють абсолютного бар'єру між їхнім визначенням та користувачем, а ґрунтуються на ввічливому принципі "невтручання у визначення". Найважливіші риси класів зберігають повну міць, але: механізм успадкування класів дозволяє численні базові класи, похідний клас може

переписати будь-який метод свого базового класу (чи класів), метод може викликати однойменний метод базового класу. Об'єкти можуть мати будь-яку кількість приватних даних.

Якщо послуговуватися термінологією мови C++, то усі члени класу (включно з членами даних) — *публічні*, а всі функції класу — *віртуальні*. Спеціальних конструкторів чи деструкторів не існує. Подібно до мови Modula-3 у Пайтоні не існує скорочень для посилання на члени об'єкта із його методів: функція методу задається із явним першим аргументом, що репрезентує об'єкт, який неявно передається при виклику цієї функції. Подібно до мови Smalltalk самі класи також є об'єктами, хоча і в ширшому сенсі: у мові Пайтон всі типи даних — об'єкти. Це визначає семантику імпортування та перейменування. На відміну від C++ та Modula-3 вбудовані типи можуть використовуватися як базові класи для створених користувачем розширень. Також, подібно до C++, але на відміну від Modula-3, більшість вбудованих операторів, що мають спеціальний синтаксис (арифметичні оператори, індексатори тощо) можуть перевизначатися в реалізаціях класів.

Контексти та іменні простори

Перед тим, як починати розповідь про класи, слід розповісти дещо про правила контексту в мові Пайтон. Визначення класів певним чином впливають на іменні простори і для того, щоб повністю розуміти, що відбувається, потрібно розуміти, як саме взаємодіють контексти та іменні простори. Знання цього матеріалу також корисне для будь-якого досвідченого програміста мови Пайтон.

Почнімо з деяких визначень.

Іменний простір — це проектування назв у об'єкти. Більшість іменних просторів імплементовані як словники мови Пайтон, що загалом зовсім непомітне (за винятком ефективності виконання програми), але це може змінитися в майбутньому. Приклади іменних просторів: вбудовані назви (функції, наприклад `abs()`, та вбудовані винятки); глобальні назви всередині модуля; локальні назви при виклику функції. В певному розумінні набір атрибутів об'єкта також утворює іменний простір. Необхідно знати, що назви із різних іменних просторів зовсім не пов'язані між собою; наприклад, два різних модулі можуть визначити функцію `maximize` без жодного ризику поплутання — при виклику потрібної функції користувачі модулів повинні префіксувати її назвою відповідного модуля.

Між іншим, слово *атрибут* тут стосується будь-якої назви, розташованої за крапкою — наприклад, у виразі `z.real`, `real` - це атрибут об'єкта `z`. Строго кажучи, посилання на назви всередині модуля — це атрибутивні посилання: у виразі `modname.funcname`, `modname` — назва об'єкта модуля і `funcname` - його атрибут. У цьому випадку відбувається пряме проектування між атрибутами модуля і визначеними в ньому глобальними назвами: вони всі поділяють один іменний простір! <footnote>За винятком одного. Об'єкти модулів мають секретний атрибут, що доступний лише для зчитування, який зветься __dict__, і який повертає словник, що використовується для імплементації іменного простору модуля; назва __dict__ — лише атрибут, який не є глобальною назвою. Очевидно його використання порушує абстракцію дизайну іменного простору і тому повинно обмежуватися лише такими речами як зневаджувач при пошуку помилок.</footnote>

Атрибути можуть бути призначені як для зчитування, так і для запису нових величин. Останнє стосується атрибутів модулів, для яких можна написати: "`modname.the_answer = 42`". Змінні атрибути можуть видалятися за допомогою твердження `del`. наприклад,"`del modname.the_answer`" видалить атрибут `the_answer` з об'єкта, на який посилається назва `modname`.

Іменні простори створюються в різний час і мають різну тривалість життя. Іменний простір вбудованих назв створюється під час завантаження інтерпретатора і ніколи не видаляється. Глобальний іменний простір модуля створюється під час зчитування його визначення і загалом існує до виходу інтерпретатора. Твердження, що виконуються інтерпретатором на найвищому рівні (зчитані зі скрипта чи введені у діалоговому режимі), вважаються частиною модуля `__main__`, і таким чином мають свій власний іменний простір. Вбудовані назви також живіть у окремому модулі, що зветься `__builtin__`.

Локальний іменний простір функції створюється під час її виклику і видаляється при поверненні нею результату або при зведенні нею винятку, необробленого у самій цій функції. (Власне термін "забування" краще відбивав би те, що відбувається насправді). І, звичайно, кожен рекурсивний виклик функції має свій власний іменний простір.

Контекст — це текстуальний регіон програми, написаної на Пайтоні, де іменний простір доступний напряму. "Доступний напряму" тут означає,

що безпосереднє посилання на назву спричинить її пошук у цьому іменному просторі.

Хоча контексти визначаються статично, вони використовуються динамічно. У будь-який момент виконання програми існують принаймні три вкладених контексти з доступними напряму іменними просторами:

1. *Внутрішній* (з якого починається пошук) містить в собі локальні назви.
2. *Іменний простір оточуючих функцій*, де пошук розпочинається з найближчого оточуючого контексту.
3. *Середній* контекст (де продовжується пошук) містить глобальні назви поточного модуля.
4. *Зовнішній контекст*, де пошук завершується, — це іменний простір, що містить вбудовані назви.

Якщо назва задекларована глобально, то всі посилання і призначення відбуваються у середньому контексті, де зберігаються глобальні назви модуля. В інакшому випадку, всі змінні, що зустрічаються поза внутрішнім контекстом, призначені лише для зчитування.

Загалом локальний контекст посилається на локальні назви (текстуально) поточної функції. Поза функціями локальний контекст посилається на той же іменний простір, що й глобальний контекст: іменний простір модуля. Визначення класів додають іще один іменний простір до локального контексту.

Важливо усвідомити, що контексти визначаються текстуально: глобальний контекст функції, визначеної в модулі, є іменним простором цього модуля, незалежно від того, звідки чи за допомогою якого псевдоніма її викликано. З іншого боку, сам пошук назв відбувається динамічно, під час виконання, хоча мова еволюціонує до статичного вирішення назв під час "компіляції", отже не варто покладатися на динамічне вирішення назв! (Уже зараз локальні змінні вирішуються статично).

Особливістю Пайтона є те, що призначення завжди відбуваються у внутрішньому контексті. Призначення не копіюють дані, а лише прив'язують назви до об'єктів. Те саме стосується і видалення: твердження "`del x`" видаляє прив'язку до `x` із іменного простору локального контексту. Насправді всі операції, що визначають нові назви, використовують локальний контекст: зокрема імпортування та визначення функцій прив'язують назву модуля чи функції у локальному

контексті. (Твердження `global` може використовуватися як індикатор того, що дана змінна існує у глобальному контексті).

Перший погляд на класи

Класи додають трохи нового синтаксису, три нових типи об'єктів та трохи нової семантики.

Синтаксис визначення класів

Найпростіша форма визначення класу виглядає так:

```
class ClassName:
    твердження-1;
    .
    .
    .
    твердження-N;
```

Визначення класів, подібно до визначень функцій (заданих через твердження `def`) , повинні виконатися перед тим, як вони матимуть якийсь ефект. (Визначення класу в принципі може розташовуватися у розгалуженні твердження `if` чи всередині функції).

На практиці твердження всередині класу здебільшого містять визначення функцій, але й інші твердження тут дозволяються також — ми повернемося до цього пізніше. Визначення функцій, розташованих всередині класів мають особливу форму списку аргументів, що диктується правилами виклику методів, але про це також йтиметься далі.

При вході у визначення класу створюється новий іменний простір, що використовується у якості локального контексту, таким чином всі призначення нових величин локальним змінним потрапляють у цей новий іменний простір. Зокрема визначення функцій прив'язують назву нової функції тут.

Після нормального виходу з визначення класу (при його закінченні) створюється *класовий об'єкт*, який фактично є оболонкою змісту іменного простору, створеного цим визначенням (про класові об'єкти йтиметься в наступному розділі). Після цього поновлюється початковий локальний контекст (той, що існував перед входом до визначення класу), у якому класовий об'єкт прив'язується до вказаної у заголовці визначення класу назви (у наведеному вище прикладі `ClassName`).

Класові об'єкти

Класові об'єкти підтримують два типи операцій: атрибутивні посилання та реалізацію.

Атрибутивні посилання використовують стандартний синтаксис атрибутивних посилань мови Пайтон у вигляді `obj.name`. Дійсні атрибутивні назви складаються з усіх назв, що існували в іменному просторі класу під час створення класового об'єкта. Тобто, якщо визначення класу виглядало таким чином:

```
class MyClass:
    "Простенький приклад класу"
    i = 12345
    def f(self):
        return 'привіт, світе'
```

то `MyClass.i` and `MyClass.f` є дійсними атрибутивними посиланнями, що повертають цілочислову величину та об'єкт методу відповідно. Атрибути класу можуть також набувати нових значень, тобто `MyClass.i` може змінюватися через призначення. `__doc__` - також дійсний атрибут; він повертає рядок документації відповідного класу: "Простенький приклад класу".

Реалізація класу використовує нотацію функцій. Просто вдайте собі, що класовий об'єкт - це безпараметрова функція, яка повертає нову реалізацію класу. Відповідно до наведеного вище прикладу

```
x = MyClass()
```

створює нову *реалізацію* класу і призначає цей новостворений об'єкт локальній змінній `x`.

Операція реалізації ("виклик" класового об'єкта) створює пустий об'єкт. Часто класи створюють об'єкти із заданим початковим станом. Для цього клас може визначити спеціальний метод, що зветься `__init__()`, наприклад:

```
def __init__(self):
        self.data = []
```

Якщо в класі визначено метод `__init__()`, то він автоматично викликається при реалізації нового класу. Для наведеного вище прикладу нова ініціалізована реалізація класу може бути отримана так:

```
x = MyClass()
```

І звичайно, метод `__init__()` для більшої зручності може мати і аргументи, що передаються йому під час реалізації класу. Наприклад:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Реалізовані об'єкти

Що ж можна робити з реалізованими об'єктами? Єдиний тип операцій, зрозумілий реалізованим об'єктам — це атрибутивне посилання. Існують два типи дійсних атрибутивних назв.

Перший тип можна умовно назвати *атрибутами* даних, *які відповідають "реалізованим змінним" у мові Smalltalk* та "членам даних" у C++. Декларувати атрибути даних непотрібно; подібно до локальних змінних вони починають існувати при першому призначенні. Наприклад, якщо `x` є реалізацією вищезгаданого класу `MyClass, то цей шматок коду виведе величину 16 і не залишить від неї жодного сліду:`

```
x.counter = 1
while x.counter &lt; 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

Другий тип атрибутивного посилання, зрозумілий реалізованим об'єктам - це *методи*. Метод — це просто функція, що "належить" об'єкту. (У Пайтоні термін "метод" вживається не лише по відношенню до

реалізацій класів: інші типи об'єктів можуть також мати методи. Зокрема, спискові об'єкти мають методи для додавання, видалення, вставки, сортування тощо. Однак у цьому розділі термін "метод" вживатиметься виняткова для позначення об'єктів реалізованих класів, якщо не зазначено інакше).

Дійсні назви методів реалізованих об'єктів залежать від відповідного класу. За визначенням усі атрибути класу, які є (визначеними користувачем) об'єктами функцій, визначають відповідні методи реалізованих об'єктів. Зокрема, `x.f` — це дійсне посилання методу, тому що `MyClass.f` — функція, але `x.i` - ні, тому що `MyClass.i` не є функцією. Але `x.f` - не те саме, що й `MyClass.f` -- це *об'єкт методу*, а не функції.

Об'єкти методів

Зазвичай виклик методу відбувається відразу:

```
x.f()
```

Для нашого прикладу це поверне рядок `'привіт, світе'`. Але виклик методу в момент посилання на нього не є обов'язковим: якщо `x.f` — об'єкт методу, то він може бути збережений і викликаний пізніше. Наприклад:

```
xf = x.f
while True:
    print xf()
```

виводитиме "`привіт, світе`" до кінця світу.

Що ж насправді відбувається при виклику методу? Можливо ви помітили, що `x.f()` було викликано без зазначеного у визначенні аргументу. Що ж трапилося? Адже відомо, що Пайтон створює виняток, коли функція, яка потребує аргументу, викликається без нього, навіть якщо той аргумент насправді не використовується...

Можливо ви вже вгадали відповідь: особливою рисою методів є те, що відповідний об'єкт передається через перший аргумент функції. Для нашого прикладу `x.f()` повністю еквівалентне `MyClass.f(x)`. Загалом виклик методу зі списком у *n* аргументів відповідає виклику

відповідної функції зі списком, що створюється додаванням об'єкта методу перед першим аргументом.

Якщо вам ще не зовсім зрозуміло, як працюють методи, висвітлення окремих деталей імплементації може дещо прояснити стан речей. При посиланні на атрибут реалізації, що не є атрибутом даних, відбувається пошук у відповідному класі. Якщо назва позначає дійсний атрибут класу, що є функційним об'єктом, то створюється об'єкт методу шляхом запаковування (пойнтера) реалізованого об'єкта та об'єкта функції в єдиному абстрактному об'єкті, що і є об'єктом методу. При виклику об'єкта методу зі списком аргументів, він знову розпаковується і новий список аргументів створюється із реалізованого об'єкта і початкового списку аргументів, після чого об'єкт функції викликається з новим списком аргументів.

Випадкові зауваження

Атрибути даних перекривають однойменні атрибути методів. Щоб запобігти випадковим конфліктам назв, що можуть призвести до проблем, які буде досить важко віднайти у великих програмах, гарною ідеєю є використання певних конвенцій, що допоможуть мінімізувати ризик таких зіткнень. Такими конвенціями може бути вживання великої літери у назвах методів, префіксація назв атрибутів даних (можливо за допомогою лише нижньої риски), використання дієслів у назвах методів та іменників у атрибутах даних тощо.

Посилання на атрибути даних може робитися як методами так і звичайними користувачами ("клієнтами") об'єкта. Іншими словами, класи не придатні для імплементації чистих абстрактних типів даних. Більше того, у Пайтоні немає нічого, що уможливлює приховування даних — все базується лише на конвенціях. (З іншого боку, імплементація Пайтона, написана на C, може повністю приховати деталі і контролювати доступ до об'єктів якщо таке потрібно; це може використовуватися розширеннями Пайтона, написаними на C).

Клієнти повинні використовувати атрибути даних обережно — бо вони можуть випадково зруйнувати незмінні величини методів, наступивши на їхні атрибути даних. Клієнти можуть також додавати власні атрибути даних до реалізації об'єкта, не порушуючи при цьому цілісності методів, якщо при цьому уникнуто конфлікту назв (знову ж таки: використання іменних конвенцій може оберегти від головного болю).

Всередині методів скорочене посилання на атрибути даних (як і на інші методи) неможливе. Як на мене, то це значно покращує читабельність методів через те, що при перегляді коду методу неможливо переплутати локальні та реалізовані змінні.

Зазвичай перший аргумент методу зветься `self`. Це лише традиція: назва `self` не має жодного значення в мові Пайтон. При цьому слід зауважити, що відступ від цієї конвенції може ускладнити читання коду іншими програмістами. Також цілком можливо, що колись буде створено спеціальний переглядач класів, що значною мірою покладатиметься на цю конвенцію.

Будь-який об'єкт функції, що є атрибутом класу, визначає метод для реалізації відповідного класу. При цьому функція не повинна знаходитися всередині цього класу текстуально: призначення об'єкта функції локальній змінній також спрацює. Наприклад:

```
# Функція, визначена поза класом
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'всім привіт!'
    h = g
```

Таким чином `f`, `g` та `h` - всі є атрибутами класу `C`, що позначають функційні об'єкти, і відповідно вони всі є методами реалізацій `C` -- при чому `h` — повністю тотожній `g`. Слід зауважити, що подібна практика зазвичай лише збиває з пантелику читача програми.

Одні методи можуть викликати один одного за допомогою атрибутів методів аргументу `self`:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
```

```
        self.add(x)
        self.add(x)
```

Методи, так само як і звичайні функції, можуть посилатися на глобальні назви. Глобальний контекст, асоційований з методом, — це модуль, де знаходиться визначення класу. (Сам же клас ніколи не використовується як глобальний контекст!). Хоча використання глобальних даних всередині методу не часто може бути виправдане, існує багато легітимних мотивів для використання глобального контексту: перш за все, функції та модулі імпортовані в глобальний контекст можуть використовуватися методами. Часто клас, що містить в собі метод, сам визначений у цьому глобальному контексті, а в наступному розділі ми побачимо й інші мотиви, через які метод потребує послання на власний клас!

Спадковість

Звичайно, без спадковості ця риса мови не могла б називатися "класом". Синтаксис похідного класу виглядає так:

```
class DerivedClassName(BaseClassName): # ПохіднийКлас(КласОснова)
    твердження-1;
    .
    .
    .
    твердження-N;
```

Назва `BaseClassName` повинна бути визначена в контексті, що містить визначення похідного класу. Замість назви класу-основи можна вживати вираз. Це корисно, коли клас визначено в іншому модулі:

```
class DerivedClassName(modname.BaseClassName):
```

Виконання визначення похідного класу відбувається так само, як і для класу-основи. При конструкції класового об'єкта клас-основа запам'ятовується. Це використовується при вирішенні атрибутивних посилань: якщо потрібний атрибут не знайдено в класі, то пошук продовжується в класі-основі. Це правило застосовується рекурсивно, якщо клас-основа є похідним від якогось іншого класу.

Реалізація похідного класу не має в собі нічого особливого: `DerivedClassName()` створює нову реалізацію класу. Посилання на методи вирішуються таким чином: відповідний класовий атрибут шукається в похідному класі і, якщо необхідно, вниз по ланцюгу класів-основ; посилання на метод є дійсним, якщо знайдено відповідний функційний об'єкт.

Похідні класи можуть перевизначити методи своїх класів-основ. Оскільки методи не мають спеціальних привілеїв при виклику інших методів свого об'єкта, метод класу-основи, що викликає інший метод, визначений у тому ж самому класі-основі, може натомість викликати метод похідного класу, що перевизначає його. (Для програмістів мови C++ усі методи Пайтона можуть вважатися віртуальними (`virtual`).

Перевизначення методу похідного класу може використовуватися не стільки для заміни, як для розширення однойменного методу. Метод класу-основи дуже просто викликати напряму: "`НазваКласуОснови.назваМетоду(посиланняНаДанийКласс, аргументи)`". Інколи це корисно і для клієнтів. (Зауважте, що це працює лише тоді, коли клас-основа визначений в глобальному контексті чи напряму імпортується в нього).

Множинна спадковість

Пайтон підтримує обмежену форму множинної спадковості. Визначення класу з кількома класами-основами виглядає так:

```
class DerivedClassName(Base1, Base2, Base3): # ПохіднийКлас(Основа1, Основа2, Основа3):
    твердження-1
    .
    .
    .
    твердження-N
```

Єдине правило, що потребує пояснення — це вирішення атрибутивних посилань. Пошук відбувається в глибину зліва направо. Тобто, якщо атрибут не знайдено в похідному класі (`DerivedClassName`), пошук продовжується у класі `Base1`, а потім (рекурсивно) в класах-основах `Base1`. В разі, якщо його тут не знайдено, пошук продовжуватиметься у `Base2` і т.д.

(Для декого пошук в ширину -- тобто пошук у `Base2` та `Base3` перед класами-основами `Base1` -- виглядає більш природнім. Однак це потребувало б знання, чи певний атрибут класу `Base1` визначений саме в ньому чи в одному з його класів-основ, перед тим як визначити наслідки конфліктів назв з атрибутами класу `Base2`. Пошук у глибину не розрізняє між прямими та успадкованими атрибутами `Base1`).

Зрозуміло, що беззастережне використання множинної спадковості може жахливо ускладнити утримання коду, особливо з огляду на те, що для уникнення конфліктів назв Пайтон покладається виключно на конвенції. Добре відома проблема множинної спадковості — коли клас походить із двох класів, які мають той самий клас-основу. Що при цьому відбувається — зрозуміти неважко (реалізація матиме єдину копію "реалізованих змінних" чи атрибутів даних, що використовуються спільним класом-основою), але не зрозуміло чи така семантика є дійсно корисною.

Приватні змінні

Існує обмежена підтримка для приватних ідентифікаторів класів. Будь-який ідентифікатор, що має форму `__spam` (мінімум дві нижні риски на початку і максимум одна в кінці) текстуально замінюється на `_classname__spam`, де `classname` - назва поточного класу, з якої було видалено початкові нижні риски. Це робиться безвідносно до синтаксичної позиції ідентифікатора, тобто він може використовуватися для визначення приватних реалізацій, класових змінних, методів та глобальних назв, навіть для зберігання реалізованих змінних, приватних для цього класу в реалізаціях *інших* класів. Скорочення може відбутися, якщо новоутворена назва довша за 255 символів. Поза класами (або якщо назва класу складається лише з нижніх рисок) жодних перетворень назви не відбувається.

Трансформація назв надає класам простий спосіб визначення "приватних" реалізованих змінних та методів без зайвих турбот щодо реалізованих змінних, визначених у похідних класах, чи зміни реалізованих змінних поза межами класу. Зауважте, що правила заміни назв створені насамперед для уникнення неприємних випадків; але при бажанні настирлива душа все одно зможе зчитати або змінити змінну, що вважається приватною. В окремих випадках (наприклад, при розробці зневаджувача) це навіть може бути корисним, що і є однією з причин чому ця дірочка ще й досі не зачинена. (Вада: при утворенні похідного

класу з тією ж назвою, що й клас-основа стає можливим доступ для приватних змінних класу-основи).

Зауважте, при передачі коду через `exec`, `eval()` чи `evalfile()` назва викликаючого класу не вважається поточною. Це подібно до твердження `global`, ефект дії якого також обмежується кодом, що був доступним у момент компіляції у послідовність байтів. Це ж обмеження стосується і `getattr()`, `setattr()` та `delattr()`, а також прямого посиланні на `__dict__`.

Різне

Інколи корисно мати тип даних подібний до "record" мови Pascal' чи "struct *мови С, що зліплює докупи кілька названих одиниць даних. Це* чудово виходить за допомогою пустого визначення класу:

```
class Employee:
    pass

john = Employee() # Створити нового робітника

# І заповнити його відповідними даними
john.name = 'Іван Іванченко'
john.dept = 'комп. лабораторія'
john.salary = 1000
```

Якщо певний код очікує особливого абстрактного типу даних, то цьому коду можна звичайно передати клас, що натомість імітує методи того типу даних. Наприклад, якщо ви маєте функцію, що форматує певні дані, отримані із файлового об'єкта, ви можете визначити клас із методами `read()` та `readline()`, що замість цього отримує дані із рядкового буфера, і передати його у якості аргументу.

Об'єкти реалізованих методів також мають атрибути: `m.im_self` - об'єкт, метод якого є реалізацією, і `m.im_func` — об'єкт функції, що відповідає цьому методу.

Винятки теж класи

Визначені користувачем винятки — також класи. Використання цього механізму дозволяє створювати розширювані ієрархії класів.

Існують дві дійсні (семантичні) форми тверджень підняття винятків:

```
raise Class, instance # raise Клас, реалізація

raise instance # raise реалізація
```

У першій формі, `instance` повинна бути реалізацією класу `Class` або іншого класу, що походить від нього. Друга форма — це скорочення для:

```
raise instance.__class__, instance
```

Клас, розташований у твердженні except, є сумісним із винятком, якщо обидва належать до одного класу, або якщо цей клас є класом-основою винятку (але не навпаки — похідний клас всередині твердження except не є сумісним із класом-основою). Зокрема, поданий нижче код виведе B, C, D у вказаному порядку:

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Зауважте, що якби твердження `except` були розташовані у зворотному порядку (з "`except B`" на початку), то вивід виглядав би B, B, B — найперше твердження `except` відфільтрувало б усі винятки.

При виводі повідомлення про помилку зсередини непозначеного винятку, що є класом, спочатку виводиться назва класу, потім двокрапка і пробіл,

і врешті-решт реалізація, переведена в рядок за допомогою вбудованої функції `str()`.

Ітератори

Ви вже напевно помітили, що більшість складених об'єктів можуть вживатися у циклічних конструкціях та оброблятися за допомогою твердження `for`:

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

Такий стиль доступу до елементів є чистим, стислим і зручним. Використання ітераторів наповнює і об'єднує мову Пайтон. Поза кулісами твердження `for` викликає функцію `iter()` для складеного об'єкта, яка повертає об'єкт-ітератор, який в свою чергу має метод `next()`, що надає послідовний доступ до елементів складеного об'єкта. Коли елементи закінчуються, `next()` відкидає виняток `StopIteration`, який повідомляє про закінчення циклу. Цей приклад ілюструє, як все це відбувається:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
    iterator object at 0x00A1DB50
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
```

```
  File "pyshell#6", line 1, in -toplevel-
    it.next()
StopIteration
```

Завдяки цьому механізму досить легко додати властивості ітератора вашим класам. Для цього слід визначити метод `__iter__()`, що повертає об'єкт з методом `next()`. Якщо сам клас визначає метод `next()`, то `__iter__()` може просто повернути `self`:

```
>>> class Reverse: # клас ЗворотнийПорядок
    "Ітератор для обробки послідовності в зворотному порядку"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for char in Reverse('spam'):
     print char

m
a
p
s
```

Генератори

Генератори — це простий і потужний інструмент для створення ітераторів. Вони виглядають як звичайні функції, але використовують твердження `yield` для повернення даних. Кожного разу коли викликається `next()`, генератор продовжує цикл там, де він востаннє зупинився (він пам'ятає всі величини, а також останнє виконане твердження). Наступний приклад показує наскільки просто можуть створюватися генератори:

```
>>> def reverse(data):
```

```
        for index in range(len(data)-1, -1, -1):
            yield data[index]

>>> for char in reverse('golf'):
        print char

f
l
o
g
```

Будь-що з того, що можна зробити з генераторами, можна також зробити і з класовими ітераторами, про які йшлося в попередньому розділі. Компактність генераторів досягається за допомогою автоматичного виклику методів `__iter__()` та `next()`.

Іншою важливою рисою є те, що локальні змінні і стани виконання автоматично зберігаються між викликами. Завдяки цьому функції виглядають набагато чистіше аніж тоді, коли використовуються класові змінні `self.index` та `self.data`.

Окрім автоматичного створення методів та збереження стану програми по закінченні циклу генератори автоматично викликають `StopIteration`. Все це разом взяте дозволяє створювати ітератори з тією ж простотою що і звичайні функції.

Генераторні вирази

Окремі прості генератори можна закодувати в короткій формі у вигляді виразів, використовуючи синтаксис, подібний до створення списків, але з напівкруглими дужками замість квадратних. Ці вирази призначені для ситуацій, коли генератори відразу використовуються оточуючою функцією. Генераторні вирази більш компактні, але менш універсальні порівняно з повними визначеннями генераторів, а також поводяться краще по відношенню до пам'яті комп'ютера аніж еквівалентні спискові конструкції.

Приклади:

```
>>> sum(i*i for i in range(10))         # сума квадратів
285

>>> xvec = [10, 20, 30]
```

```
>>> yvec = [7, 5, 3]
>>> sum(x'y for x,y in zip(xvec, yvec)) # скалярний
добуток
260

>>> from math import pi, sin
>>> # таблиця синусів:
>>> sine_table = dict((x, sin(x'pi/180)) for x in
range(0, 91))
>>> # список словоформ:
>>> unique_words = set(word  for line in page  for word
in line.split())

>>> # перший з найкращих:
>>> valedictorian = max((student.gpa, student.name) for
student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1,-1,-1))
['f', 'l', 'o', 'g']
```

Інтерфейс операційної системи

Модуль `os` містить функції взаємодії з операційною системою:

```
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd() # Повертає поточну робочу директорію
'C:\\Python24'
>>> os.chdir('/server/accesslogs')
```

Пам'ятайте, що слід використовувати "`import os`" замість "`from os import *`". Це дозволить запобігти перекриванню вбудовано функції `open()` функцією `os.open()`, яка має зовсім інше призначення.

Вбудовані функції `dir()` та `help()` є дуже корисними для отримання допомоги при роботі з такими великими модулями як `os`:

```
>>> import os
>>> dir(os) # повертає список усіх функцій модуля
```

```
>>> help(os) # повертає інструкцію створену збиранням до купи рядків документації модуля
```

Для щоденних потреб, пов'язаних з файлами та директоріями, модуль `shutil` надає інтерфейс більш високого рівня, що спрощує програмування:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db') # копіювання
>>> shutil.move('/build/executables', 'installdir') # переміщення
```

Шаблони розширення файлових назв

Модуль `glob` містить функцію, що дозволяє створювати списки файлів за допомогою шаблонів розширення, застосованих до директорій:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

Аргументи командного рядка

Скрипти часто використовують аргументи, подані з командного рядка. Ці аргументи зберігаються у вигляді списку атрибута *argv*, що знаходиться в модулі `sys`. Наприклад, якщо з командного рядка було запущено команду "`python demo.py one two three`", то ми можемо отримати такий вивід:

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

Модуль `getopt` оброблює *sys.argv* на основі конвенцій юніксової функції `getopt()`. Потужнішу і гнучкішу обробку командного рядка можна знайти у модулі `optparse`.

Переспрямування виводу помилок та вихід із програми

Модуль `sys` має також атрибути *stdin*, *stdout* та *stderr* ("стандартний ввід", "стандартний вивід" та "стандартний вивід помилок" відповідно). Останній корисний для виводу попереджень і помилок при переспрямуванні *stdout*:

```
>>> sys.stderr.write('Попередження: файл для запису не знайдено; створюється новий файл')
Попередження: файл для запису не знайдено; створюється новий файл
```

Найпростіший шлях виходу з програми — це виклик "`sys.exit()`".

Пошук за шаблоном

Модуль `re` містить утиліти регулярних виразів для пошуку за шаблоном всередині рядків. Регулярні вирази надають компактні оптимальні вирішення при застосуванні доволі складних правил пошуку:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Якщо потрібні лише простенькі маніпуляції, то найліпше застосовувати методи рядків, які набагато простіше читати та зневаджувати:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

Математика

Модуль `math` надає можливість доступу до функцій бібліотеки C для роботи з дробовими числами:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
```

```
>>> math.log(1024, 2)
10.0
```

Модуль `random` містить утиліти для роботи з випадковими числами:

```
>>> import random
>>> print random.choice(['яблуко', 'груша', 'банан'])
'яблуко'
>>> random.sample(xrange(100), 10)   # вибір без
заміщення
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # випадкове число з рухомою комою
0.17970987693706186
>>> random.randrange(6)    # випадкове ціле число,
вибране з послідовності range(6)
4
```

Доступ до інтернету

Існують кілька модулів для доступу до інтернету та обробки його протоколів. Два найпростіші — це `urllib2` (для отримання даних з інтернет-адрес) та `smtplib` для відправлення електронної пошти:

```
>>> import urllib2
>>> for line in
urllib2.urlopen('http://tycho.usno.navy.mil/cgi-
bin/timer.pl'):
... if 'EST' in line:      # шукаємо Eastern Standard
Time
...     print line

Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@tmp.org',
'jceasar@tmp.org',
"""To: jceasar@tmp.org
From: soothsayer@tmp.org

Beware the Ides of March.
```

```
"""")
>>> server.quit()
```

Час і число

Модуль `datetime` містить класи для роботи з даними, що виражають час та число, як у складний так і в простий спосіб. Він придатний і для арифметики часових даних, хоча основна увага приділяється ефективному витягу даних для форматування та їхньої обробки. Модуль також має об'єкти, що розрізняють різні часові зони.

```
# створення та форматування чисел дуже просте
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y or %d%b %Y is a %A on the %d day of %B")
'12-02-03 or 02Dec 2003 is a Tuesday on the 02 day of December'

# часові дані придатні для застосування календарної арифметики
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

Ущільнення даних

Поширені формати ущільнення та архівації даних напряму підтримуються такими модулями
як `zlib`, `gzip`, `bz2`, `zipfile` та `tarfile`.

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
```

```
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(t)
-1438085031
```

Обчислення продуктивності[ред.]

Окремі користувачі мови Пайтон мають неабияку зацікавленість у тому, наскільки продуктивними є різні підходи вирішення однієї проблеми відносно один одного. Пайтон має інструменти, що дозволяють швидко віднайти відповіді на ці питання.

Наприклад, можливо комусь захотілося використовувати кортежі замість більш традиційного способу обміну величин двох змінних.
Модуль `timeit` швидко демонструє, що традиційний спосіб є набагато швидшим:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.60864915603680925
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.8625194857439773
```

На відміну від високого рівня детальності модуля `timeit`, модулі `profile` та `pstats` мають інструменти для ідентифікації критичних ділянок коду у більших блоках коду.

Контроль якості

Один із способів для створення якісного програмного забезпечення - це написання спеціальних тестів для кожної функції, і часте використання цих тестів під час процесу розробки.

Модуль `doctest` має спеціальні інструменти для сканування модуля та перевірки тестів, що вказані в рядках документації. Створення ж тестів — дуже просте і полягає у копіюванні та вставці типового виклику функції та її результату в рядок документації. Додання прикладу вдосконалює документацію а також дозволяє модулю doctest перевірити, чи відповідає код документації:

```
def average(values):
```

```
    """Виводить середнє арифметичне для даного списку
чисел.

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod()    # автоматично перевірити тести
```

Модуль `unittest` є дещо складнішим за `doctest`, але натомість дозволяє провести більш ґрунтовне тестування за допомогою правил, що здебільшого задаються в окремому файлі:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1),
4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main() # Виклик з командного рядка запускає всі
тести
```

Батарейки додаються

Коротко філософію Пайтона можна висловити так: "батарейки додаються". Це найкраще видно через складні та потужні властивості більших пакетів. Зокрема:

- Модулі `xmlrpclib` та `SimpleXMLRPCServer` роблять розробку викликів віддалених процедур досить тривіальною справою. Не зважаючи на назви, їхнє використання не потребує спеціальних навичок роботи з XML.
- Пакет `email` — це бібліотека для роботи з електронними повідомленнями, що включає MIME та інші документи повідомлень,

які визначаються у RFC 2822. На відміну від `smtplib` та `poplib`, які просто надсилають та отримують повідомлення, цей пакет має повний інструментарій для створення та кодування складних структур повідомлень (включно з доданими документами) та для імплементації інтернет-кодувань та протоколів заголовків.

- Пакети `xml.dom` та `xml.sax` мають потужну підтримку для обробки цього популярного формату для обміну даними. Подібно до нього модуль `csv` підтримує зчитування і запис розповсюдженого формату бази даних. Разом ці модулі та пакети значно спрощують обмін даними між Пайтоном та іншими програмами та утилітами.
- Кілька модулів присвячено інтернаціоналізації, зокрема `gettext`, `locale` та `codecs`.

- Форматування виводу
- Модуль `repr` має версію функції `repr()` для скороченого зображення великих або багаторівневих структур даних:

```
>>> import repr
>>>
repr.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

- Модуль `pprint` (pretty printer) надає можливість більш досконалого контролю при виводі об'єктів (як вбудованих, так і заданих користувачем) у вигляді, придатному для зчитування інтерпретатором. Якщо результат довший за один рядок, то ця функція додає пробіли та символи нового рядка, які дозволяють ясніше виразити структуру даних:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green',
'red']], [['magenta',
...       'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

- Модуль `textwrap` форматує текст для певної ширини екрану:

```
>>> import textwrap
>>> doc = """Метод wrap() подібний до fill(), але він повертає
... список рядків замість одного довгого рядка, розбитого
... на рядки."""
...
>>> print textwrap.fill(doc, width=40)
Метод wrap() подібний до fill(),
але він повертає список рядків
замість одного довгого рядка,
розбитого на рядки.
```

- Модуль `locale` завантажує формати даних, специфічні для певного культурного оточення. Спеціальний атрибут форматуючої функції модуля надає можливість прямого форматування чисел за допомогою групових роздільників:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'uk_UA.utf8')
('uk_UA', 'utf8')
>>> conv = locale.localeconv()    # отримати правила переведення
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1.234.567'
>>>  print locale.format("%.*f%s",
...           (conv['int_frac_digits'], x,
...           conv['currency_symbol']), grouping=True)
1.234.567,80гр
```

- Шаблони

- Модуль `string` має зручний клас `Template` з досить простим синтаксисом, придатним для редагування користувачами. За його допомогою користувачі можуть змінювати текстові величини програми без внесення змін до її коду.
- Формат модуля використовує спеціальні назви-заповнювачі, що утворюються за допомогою символа "`$`" та дійсного ідентифікатора мови Пайтон (буквено-цифрові символи та нижня риска). Для відокремлення назви-заповнювача від наступних буквено-

цифрових символів слід використовувати фігурні дужки. "`$$`" задає один символ "`$`":

```
>>> from string import Template
>>> t = Template('${village} витратили $$10 на $cause.')
>>> print t.substitute(village='Васюки', cause='сміттєфонд')
Васюки витратили $10 на сміттєфонд.
```

- Метод `substitute` відкидає `KeyError` якщо ключове слово не існує в словнику або в ключовому аргументі. Для програм, де дані для заповнення шаблонів можуть бути неповними, краще використовувати метод `safe_substitute`, що за умови відсутності відповідних даних залишить незаповнені назви без змін.

```
>>> t = Template('Повернути $item $owner.')
>>> d = dict(item='непроковтнутий шматок')
>>> t.substitute(d)
Traceback (most recent call last):
  . . .
KeyError: 'owner'
>>> print t.safe_substitute(d)
Повернути непроковтнутий шматок $owner.
```

- Похідні класи-шаблони можуть визначати свої власні роздільники. Наприклад, спеціальний утиліт для зміни назви групи файлів, що є частиною фото-переглядача, може використовувати символ процента для позначення таких назв-заповнювачів як, скажімо, число, номер послідовності чи формат файла:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = raw_input('Введіть формат зміни назви (%d-число %n-номер %f-формат):  ')
Введіть формат зміни назви (%d-число %n-номер %f-формат):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
```

```
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print '%s --&gt; %s' % (filename, newname)

img_1074.jpg --&gt; Ashley_0.jpg
img_1076.jpg --&gt; Ashley_1.jpg
img_1077.jpg --&gt; Ashley_2.jpg
```

- Інше можливе використання шаблонів — це відокремлення логіки програми від деталей численних форматів виводу (наприклад, простого тексту, XML, HMTL тощо).

Робота з форматами бінарних записів даних

- Модуль `struct` має функції `pack()` та `unpack()`, що призначені для роботи з бінарними записами різної величини. Наступний приклад показує, як можна зчитати інформацію із заголовка файла у форматі ZIP (коди пакування `"H"` та `"L"` відповідно виражають три- та чотирибайтові беззнакові числа):

```
    import struct

    data = open('myfile.zip', 'rb').read()
    start = 0
    for i in range(3):      # три перші заголовки файла
        start += 14
        fields = struct.unpack('LLLHH', data[start:start+16])
        crc32, comp_size, uncomp_size, filenamesize, extra_size =  fields

        start += 16
        filename = data[start:start+filenamesize]
        start += filenamesize
        extra = data[start:start+extra_size]
        print filename, hex(crc32), comp_size, uncomp_size

        start += extra_size + comp_size     # перейти до іншого заголовка
```

Розгалуження

- Розгалуження (threading) — це технологія відокремлення послідовно незалежних задач. Галузки можуть використовуватися для покращення ефективності програм, що отримують дані від користувача під час виконання інших задач. Ще одне можливе застосування — здійснення вводу-виводу під час паралельного обчислення іншою галузкою.
- Наступний код ілюструє, як модуль високого рівня `threading` виконує фонові задачі під час виконання головної програми.

```
    import threading, zipfile

    class AsyncZip(threading.Thread): # асинхронне ущільнення
        def __init__(self, infile, outfile):
            threading.Thread.__init__(self)
            self.infile = infile
            self.outfile = outfile
        def run(self):
            f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
            f.write(self.infile)
            f.close()
            print 'Закінчено фонове ущільнення файла ', self.infile

    background = AsyncZip('mydata.txt', 'myarchive.zip')
    background.start()
    print 'Головна програма продовжує виконуватися на передньому фоні.'

    background.join()    # Чекаємо закінчення виконання фонової задачі
    print 'Головна програма дочекалася завершення фонової задачі.'
```

- Головна складність при розгалуженні програми полягає в координації окремих галузок, що поділяють дані чи інші ресурси. Для цього модуль threading має кілька примітивних функцій синхронізації, що включають замки, події, умовні змінні та семафори.
- Ці інструменти дуже потужні, але навіть невеликі помилки дизайну можуть призвести до проблем, які згодом буде важко відтворити. Таким

чином, найкращим способом для координації задач є зосередження доступу до ресурсів у єдиній галузці з подальшим використанням модуля Queue для передачі цій галузці запитів від інших галузок. Програми, що використовують для координації і комунікації між галузками об'єкти типу Queue, відзначаються відносною простотою дизайну, читабельністю та надійністю.

- Журнальні записи
- Модуль logging пропонує гнучку і потужну систему для створення журнальних записів. У найпростішому випадку журнальні повідомлення виводяться у файл чи на стандартний вивід для запису помилок (sys.stderr):

```
import logging
logging.debug('Інформація по зневадженню')
logging.info('Інформаційне повідомлення')
logging.warning('Попередження: конфігураційний файл %s не знайдено', 'server.conf')
logging.error('Відбулася помилка')
logging.critical('Критична помилка -- програма зачиняється')
```

- Це видає такий вивід:

```
WARNING:root:Попередження: конфігураційний файл server.conf не знайдено
ERROR:root:Відбулася помилка
CRITICAL:root:Критична помилка -- програма зачиняється
```

- Загалом інформаційні повідомлення та повідомлення по налагодженню замовчуються, а вивід здійснюється на стандартний потік для виводу помилок. Інші можливості виводу включають передачу повідомлень через електронну пошту, датаграми, сокети або HTTP-сервер. Повідомлення можуть отримувати різні маршрути в залежності від пріоритету повідомлень: DEBUG (зневадження), INFO (загальна інформація), WARNING (попередження), ERROR (помилка) та CRITICAL (критичне повідомлення).
- Система ведення журнальних записів може бути налаштована напряму з Пайтона або завантажена з відредагованого користувачем конфігураційного файла без внесення жодних змін до існуючих програм.
- Слабкі посилання

- Пайтон автоматично керує пам'яттю за допомогою підрахунку посилань для більшості об'єктів та автоматичного смітника для видалення циклів. Пам'ять автоматично звільняється незабаром після видалення останнього посилання.
- Такий підхід задовольняє більшість програм, але інколи потрібно утримувати об'єкти лише тоді, коли вони десь використовуються. На жаль, створення ще одного посилання робить ці об'єкти постійними. Модуль `weakref` надає можливість відстежувати об'єкти без створення посилань. Коли об'єкт більше не потрібен, він автоматично видаляється із таблиці слабких посилань (при цьому викликається спеціальна функція для об'єктів, що є слабкими посиланнями). Типове використання слабких посилань — це зберігання в пам'яті об'єктів, створення яких потребує значних ресурсів:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...             self.value = value
...     def __repr__(self):
...             return str(self.value)
...
>>> a = A(10)                    # створюємо посилання
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a             # посилання не створюється
>>> d['primary']                 # дістає об'єкт, якщо він ще існує
10
>>> del a                        # видаляємо посилання
>>> gc.collect()                 # запускаємо сміттярку
0
>>> d['primary']                 # елемент автоматично видаляється
Traceback (most recent call last):
  File "pyshell#108", line 1, in -toplevel-
    d['primary']                # елемент автоматично видаляється
  File "C:/PY24/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
```

```
    KeyError: 'primary'
```

- Знаряддя для роботи зі списками
- Вбудовані списки можуть використовуватися у різноманітних структурах даних. Але інколи потрібно застосовувати альтернативні рішення, які можуть вплинути на ефективність виконання програми.
- Модуль `array` містить об'єкт `array()` для створення масивів, що подібний до списку, який містить лише однорідні дані, і який зберігає їх більш компактно. Подальший приклад ілюструє масив чисел, що зберігається у вигляді двобайтових беззнакових бінарних чисел (код `"H"`) замість звичайних 16-байтових елементів у списках цілочислових величин мови Пайтон.

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

- Модуль `collections` містить об'єкт `deque`, (double-ended queue — "черга з двома кінцями"), що подібний до списку з пришвидшеним видаленням і додаванням до його лівої частини, але повільнішим пошуком елементів, розташованих посередині. Ці об'єкти призначені для створення черг та дерев, де пошук відбувається спочатку в ширину, а потім в глибину.

```
>>> from collections import deque
>>> d = deque(["задача1", "задача2", "задача3"])
>>> d.append("задача4")
>>> print "Опрацьовується", d.popleft()
Опрацьовується задача1

# пошук в ширину
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

- Окрім альтернативних імплементацій списків, існує також модуль `bisect`, що містить функції для роботи з упорядкованими списками:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

- Модуль `heapq` містить функції для створення стосів (heaps) на основі звичайних списків. Елемент, що має найменшу величину, завжди знаходиться в нульовій позиції. Це корисно, коли потрібен постійний доступ до найменшого елемента, але при цьому ви не хочете сортувати весь список.

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # пересортувати список у стос
>>> heappush(data, -5)                 # додати новий елемент
>>> [heappop(data) for i in range(3)]  # дістати три найменші елементи
[-5, 0, 1]
```

- Арифметика десяткових чисел з рухомою комою

- Модуль `decimal` містить десятковий тип даних, що зветься `Decimal`, призначений для роботи з десятковими дробами. Порівняно із вбудованою імплементацією двійкових дробів (на основі типу `float`), цей новий клас особливо корисний при розробці фінансових та інших програм, що потребують точної десяткової репрезентації, контролю округлення для задоволення різних правових чи цивільних вимог, збереження значущих десяткових знаків, або тоді, коли результати комп'ютерного обчислення повинні відповідати підрахунку, зробленому на папері.
- Наприклад, підрахунок п'ятивідсоткового податку на телефонну розмову вартістю 70 копійок дає відмінні результати при використанні двійкових та десяткових дробів. Ця різниця стає істотною при округленні до однієї копійки.

```
>>> from decimal import *
>>> Decimal('0.70') * Decimal('1.05')
Decimal("0.7350")
>>> .70 * 1.05
0.73499999999999999
```

- Результат цього обчислення, що належить до типу `Decimal`, утримує кінцевий нуль, автоматично задаючи чотири значущих десяткові знаки на підставі множення двох чисел з двома десятковими знаками. Десяткові числа відтворюють результати отримані на папері, що дозволяє уникнути випадків, коли двійкові числа не можуть точно виразити десяткові величини.
- Точна репрезентація за допомогою класу `Decimal` дозволяє безпомилково обчислити залишок чи перевірити рівність величин, що неможливо зробити при використанні двійкових дробів:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal("0.00")
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

- Модуль `decimal` може здійснювати арифметичні дії з будь-якою заданою точністю:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal("0.142857142857142857142857142857142857")
```

ЛАБОРАТОРНА РОБОТА №1.
ОСНОВИ МОВИ PYTHON

Мета роботи: ознайомитися з алгоритмами послідовної (лінійної) структури, з процедурами запуску програм, які реалізують ці алгоритми на мові Python; знайомство з інтегрованим середовищем розробки – integrated development environment (IDLE).

Хід роботи:

Завдання 1. Створіть чотири змінні. За допомогою функції ***input*** присвойте змінним значення з цілих і дробових чисел.

Завдання 2. Виконайте над числами наступні дії:

- додавання
- віднімання
- множення
- ділення
- піднесення до ступеня
- цілочисленне ділення
- остача від ділення двох чисел

Отримані відповіді запишіть в список.

Завдання 3. Визначте кількість елементів у попередньому списку. Виведіть на екран парні елементи списку.

Завдання 4. Поміняйте місцями другий і п’ятий елементи попереднього списку. Виведіть на екран отриманий список.

Завдання 5. Створіть змінну **name** і за допомогою функції ***input*** присвойте їй в якості значення ваше прізвище та ім’я. Виведіть на екран повідомлення про виконавця даної лабораторної роботи і висновки по ній. Речення виводяться пострічково.

Звіт по лабораторній роботі має містити:
- лістинг програми з коментарями;
- висновки по проведеній роботі.

ЛАБОРАТОРНА РОБОТА №2.
ВКАЗІВКА РОЗГАЛУЖЕННЯ

Мета роботи: Мета роботи: познайомитися із структурою розгалуження (if, if-else, if-elif-else). Навчитися працювати з числами і рядками використовуючи дану структуру..

Хід роботи:

Завдання 1. Дано три цілих числа. Вибрати з них ті, які належать інтервалу [1,3].

Завдання 2. Дано номер року (позитивне ціле число). Визначити кількість днів в цьому році, враховуючи, що звичайний рік нараховує 365 днів, а високосний - 366 днів. Високосним вважається рік, ділиться на 4, за винятком тих років, які діляться на 100 і не діляться на 400 (наприклад, роки 300 1300 і 1900 не є високосними, а 1200 і 2000 - є).

Завдання 3. Написати програму обчислення вартості покупки з урахуванням знижки. Знижка в 3% надається в тому випадку, якщо сума покупки більше 500 руб., В 5% - якщо сума більше 1000 руб.

Завдання 4. Знайти косинус мінімального з 4 заданих чисел.

Завдання 5. Вивести на екран синус максимального з 3 заданих чисел.

Завдання 6. Складіть програму підрахунку площі рівнобедреного трикутника. Якщо площа трикутника парна, розділити її на 2, в іншому випадку вивести повідомлення «Не можу ділити на 2!»

Завдання 7. Скласти програму, яка по даному числу (1-12) виводить назву відповідного йому місяця англійською мовою.

Завдання 8. Дано три числа. Знайти кількість позитивних чисел серед них;

Звіт по лабораторній роботі має містити:
- лістинг програми з коментарями;
- висновки по проведеній роботі.

ЛАБОРАТОРНА РОБОТА №3.
РОБОТА З ЦИКЛАМИ

Мета роботи: ознайомитися з циклічними конструкціями і їх використанням в мові Python.

Хід роботи:

Завдання 1. Дано два числа A і B (A <B). Знайти суму всіх цілих чисел від A до B включно. Вирішити задачу використовуючи циклічну конструкцію while.

Завдання 2. Дано два числа A і B (A <B). Знайти суму квадратів всіх цілих чисел від A до B включно. Вирішити задачу використовуючи циклічну конструкцію while.

Завдання 3. Знайти середнє арифметичне всіх цілих чисел від a до b (значення a і b вводяться з клавіатури; b≤ 200). Вирішити задачу використовуючи циклічну конструкцію while.

Завдання 4. Знайти суму всіх цілих чисел від a до b (значення a і b вводяться з клавіатури; b ≥a). Вирішити задачу використовуючи циклічну конструкцію for.

Завдання 5. Знайти суму квадратів всіх цілих чисел від a до 50 (значення a вводиться з клавіатури; 0 ≤a≤50). Вирішити задачу використовуючи циклічну конструкцію for.

Завдання 6. Дано ціле число N (> 1). Знайти найменше ціле число K, при якому виконується нерівність 5 ^ K> N. Вирішити задачу використовуючи циклічну конструкцію while.

Завдання 7. Серед чисел 1, 4, 9, 16, 25, ... знайти перше число, більше n. Вирішити задачу використовуючи циклічну конструкцію for.

Завдання 8. Серед чисел 1 5 10, 17, 26, ... знайти перше число, більше n. Умовний оператор не використовувати. Вирішити задачу використовуючи циклічну конструкцію while.

Звіт по лабораторній роботі має містити:
- лістинг програми з коментарями;
- висновки по проведеній роботі.

ЛАБОРАТОРНА РОБОТА №4.
РЯДКИ

Мета роботи: ознайомитися зі рядками в мові Python, діями над ними.

Хід роботи:

Завдання 1. Дано рядок, що містить текст (до тисячі слів). Знайти кількість слів, що починаються з заданої користувачем літери без врахування регістру.

Завдання 2. В тексті замінити всі двокрапки (:) знаком відсотка (%). Підрахувати кількість замін.

Завдання 3. В тексті видалити символ крапки (.) І підрахувати кількість виділених символів.

Завдання 4. В тексті замінити букву (а) буквою (о). Підрахувати кількість замін. Підрахувати, скільки символів в рядку.

Завдання 5. У рядку замінити всі великі літери малими.

Завдання 6. У рядку видалити всі літери "о" і підрахувати кількість віддалених символів.

Завдання 7. Дано рядок. Перетворити його, замінивши зірочками всі букви "п", що зустрічаються серед перших n / 2 символів. Тут n - довжина рядка.

Завдання 8. Визначити, скільки разів в тексті зустрічається задане слово.

Завдання 9. Дано рядок - речення на англійській мові. Перетворити рядок так, щоб кожне слово починалося з великої літери.

Завдання 10. Дано рядок. Вивести всі слова, що починаються на літеру N і слова що закінчуються на літеру P. Літери N і P вводяться користувачем.

Звіт по лабораторній роботі має містити:
- лістинг програми з коментарями;
- висновки по проведеній роботі.

ЛАБОРАТОРНА РОБОТА №5.
РОБОТА ЗІ СПИСКАМИ

Мета роботи: ознайомитися методами роботи зі списками в мові Python.

Хід роботи:

Завдання 1. Дано одновимірний масив, що складається з N цілочисельних елементів. Ввести масив з клавіатури. Знайти максимальний елемент. Вивести масив на екран у зворотному порядку.

Завдання 2. Дано одновимірний масив, що складається з N цілочисельних елементів. Ввести масив з клавіатури. Переписати всі додатні елементи в другій масив, а решту - в третій.

Завдання 3. В одновимірному числовому масиві D довжиною n обчислити суму елементів з непарними індексами. Вивести на екран масив D, отриману суму.

Завдання 4. Дано масив цілих чисел. Знайти максимальний елемент масиву і його порядковий номер. Отримати інший масив, що складається тільки з непарних чисел вихідного масиву або повідомити, що таких чисел немає. Отриманий масив вивести в порядку зменшення елементів.

Завдання 5. Дано одновимірний масив з 10 цілих чисел. Вивести пари від’ємних чисел, що стоять поруч.

Завдання 6. Дано одновимірний масив з 10 цілих чисел. Знайти максимальний елемент і порівняти з ним інші елементи. Квадрати менших чисел записати в другий масив в порядку зменшення.

Завдання 7. Дано одновимірний масив, що складається з N елементів. Ввести масив з клавіатури. Знайти та вивести мінімальний по модулю елемент. Вивести масив на екран у зворотному порядку.

Звіт по лабораторній роботі має містити:
- лістинг програми з коментарями;
- висновки по проведеній роботі.

ЛАБОРАТОРНА РОБОТА №6.
ФУНКЦІЇ

Мета роботи: ознайомитися основами функціонального програмування і використання користувацьких функцій в мові Python

Хід роботи:

Завдання 1. Користувач вводить дві сторони трьох прямокутників. Вивести їх площі.

Завдання 2. Дано катети двох прямокутних трикутників. Написати функцію обчислення довжини гіпотенузи цих трикутників. Порівняти і вивести яка з гіпотенуз більше, а яка менше.

Завдання 3. Задано коло (x-a) 2 + (y-b) 2 = R2 і точки P (p1, p2), F (f1, f1), L (l1, l2). З'ясувати і вивести на екран, скільки точок лежить всередині кола. Перевірку, чи лежить точка всередині кола, оформити у вигляді функції.

Завдання 4. Дано числа X, Y, Z, T - довжини сторін чотирикутника. Обчислити його площу, якщо кут між сторонами довжиною X і Y - прямий.

Завдання 5. Знайти всі натуральні числа, що не перевищують заданого n, які діляться на кожне із заданих користувачем чисел.

Завдання 6. Скласти програму для знаходження чисел з інтервалу [M, N], що мають найбільшу кількість дільників.

Звіт по лабораторній роботі має містити:
- лістинг програми з коментарями;
- висновки по проведеній роботі.

ЛАБОРАТОРНА РОБОТА №7.
РОБОТА З ФАЙЛАМИ

Мета роботи: ознайомитися з засобами роботи з файлами в мові Python, діями над ними.

Хід роботи:

Завдання 1. Створіть новий файл numbers.txt у текстовому редакторі і запишіть у нього 10 чисел, кожне з нового рядка. Напишіть програму, яка зчитує ці числа з файла і обчислює їх суму, виводить цю суму на екран і, водночас, записує цю суму у інший файл з назвою sum_numbers.txt.

Завдання 2. Реалізуйте програму, яка зчитує ціле число, що вводиться з командного рядка, і записує у текстовий файл інформацію, щодо парності або непарності числа.

Завдання 3. Створіть новий файл у текстовому редакторі і напишіть кілька рядків тексту у ньому про можливості Python. Кожен рядок повинен починатися з фрази: In Python you can .... Збережіть файл з ім’ям learning_python.txt. Напишіть програму, яка зчитує файл і виводить текст з перебором рядків об’єкта файла і зі збереженням рядків у списку з подальшим виведенням списку поза блоком with**.**

Завдання 4. Функція replace() може використовуватися для заміни будь-якого слова у рядку іншим словом. Прочитайте кожен рядок зі створеного у попередньому завданні файла learning_python.txt і замініть слово Python назвою іншої мови, наприклад C при виведенні на екран.

Завдання 5. Створіть порожній файл guest_book.txt у текстовому редакторі. Напишіть цикл while, який запитує у користувачів імена. При введенні кожного імені виведіть на екран рядок з вітанням для користувача і запишіть рядок вітання у файл з ім’ям guest_book.txt. Простежте за тим, щоб кожне повідомлення розміщувалося в окремому рядку файла. Передбачте вихід з циклу.

Звіт по лабораторній роботі має містити:
- лістинг програми з коментарями;
- висновки по проведеній роботі.

ЛАБОРАТОРНА РОБОТА №8.

БАЗИ ДАНИХ

Мета роботи: ознайомитися з роботою з базами даних в мові Python

Хід роботи:

Завдання 1. Використайте модуль sqlite3, щоб створити базу даних SQLite під назвою cs.db і таблицю ratings, що містить наступні поля:
id INTEGER PRIMARY KEY, firstname VARCHAR(20),
lastname VARCHAR(20), lab1 INT, lab2 INT, lab3 INT, lab4 INT, lab5 INT, lab6 INT, lab7 INT, lab8 INT, test1 INT, test2 INT, test3 INT, test4 INT, rating FLOAT.

Завдання 2. Заповніть таблицю згідно балів, вказаних в журналі оцінювання вашої групи. Поле rating обчислюється за формулою

(lab1+lab2+lab3+lab4+lab5+lab6+lab7+lab8)*1,5+(test1+test2+test3+test4)/4

Завдання 3. Зчитайте і виведіть на екран усі значення таблиці ratings у алфавітному порядку за полем firsname.

Завдання 4. Зчитайте і виведіть на екран прізвища та імена студентів з підсумковою оцінкою більшою за 60, вказавши яку оцінку за шкалою ECTS отримає студнент. Шкала ECTS вказана на рис. 1

За шкалою ECTS	За національною шкалою		За шкалою ЖДТУ (в балах)
	іспит	залік	
A	відмінно	зараховано	90 – 100
B	добре		82 – 89
C			74 – 81
D	задовільно		64 – 73
E			60 – 63
FX	незадовільно	незараховано	35 – 59
F			1 – 34

Завдання 5. Зчитайте і виведіть на екран прізвища та імена студентів з підсумковою оцінкою меншою за 50. Розрахуйте і виведіть на екран скільки балів їм потрібно набрати, щоб отримати допуск до іспиту.

Звіт по лабораторній роботі має містити:
- лістинг програми з коментарями;
- висновки по проведеній роботі.