Східноєвропейський національний університет імені Лесі Українки

Кафедра експериментальної фізики та
інформаційно-вимірювальних технологій

Замруєва О. В., Кримусь А. С., Ольхова Н. В.

Об’єктно-орієнтоване програмування в Python

Курс лекцій

Луцьк
Вежа-Друк
2018

УДК 629.33:004.94(07)
ББК 39.33:30.2я7
З 10

Рекомендовано до друку науково-методичною радою Східноєвропейського національного університету імені Лесі Українки (протокол № 6 від 21 березня 2018 р.).

Рецензенти:
Сахнюк В. Є. – кандидат фізико-математичних наук, доцент кафедри теоретичної та математичної фізики, СНУ імені Лесі Українки;
Івановський Ю.В. – DevOps інженер міжнародної корпорації «Appodeal», США.

З Замруєва О. В., Кримусь А. С., Ольхова Н. В. **Об'єктно-орієнтоване програмування в Python :** курс лекцій. Луцьк : Вежа-Друк, 2018. – 64 с.

Курс лекцій призначений для вивчення студентами ВНЗ мови програмування Python. Інтерпретована об'єктно-орієнтована мова програмування Python відноситься до тих рідкісних мов, які можуть бути визнані простими і в той же час потужними.

Python – це впешу чергу, зручна мова програмування, яка не потребує багато лишніх кодів. Однак за свою локанічність Python платить швидкістю виконання і обсягами пам'яті.

Метою курсу є встановлення загальних уявлень про мову програмування високого рівня: способах трансляції програмного коду; типів даних (цілі числа, числа з плаваючою точкою, рядки) і структури даних (рядки, списки, словники), змінних; виразів; розгалужень (`if`, `if-else`, `if-elif-else`) та циклів (`while`, `for`); введення і виведення даних; поняття про функції, локальних і глобальних змінних.

Структура видання охоплює навчальні дисципліни «Комп'ютерні програми», «Основи технічної інформатики», «Технічне проектування», «Програмування та наукові розрахунки на мові Phyton» й містять набір матеріалів необхідних для організації повноцінної аудиторної та самостійної роботи студентів.

Навчальне видання відповідає чинним навчальним програмам підготовки й рекомендовано студентам спеціальностей 014 Середня освіта (фізика), 104 Фізика та астрономія, 105 Прикладна фізика та наноматеріали, 151 Автоматизація та комп'ютерно-орієнтовані технології.

УДК 629.33:004.94(07)
ББК 39.33:30.2я7

ЗМІСТ

ВСТУП

Python – одна з тих рідкісних мов програмування, які одночасно претендують на звання простих і потужних. Вас приємно здивує те, як легко можна зосередитися на вирішенні поставленого завдання, а не на синтаксисі і структурі мови.

Python – це проста в освоєнні і потужна мова програмування. Вона надає ефективні високорівневі структури даних, а також простий, але ефективний підхід до об'єктно-орієнтованого програмування. Елегантний синтаксис і поєднання динамічної типізації з інтерпретацією, роблять Python ідеальною мовою для написання програм та швидкої розробки додатків в різних областях на більшості платформ.

Творець мови Python – Гвідо ван Россум. Назвав його так на честь телешоу на BBC під назвою «Monty Python's Flying Circus».

РОЗДІЛ 1. ОСОБЛИВОСТІ PYTHON

Програму можна представити як набір послідовних команд (алгоритмів) об’єкту, який повинен їх виконати для досягнення поставленої задачі.

Python – це інтерпретована мова програмування, вихідний код якої, частинами перетворюється в машинний у процесі виконання спеціальної програми – інтерпретатора.

Python характеризується конкретним логічним синтаксисом. Читати код на цій мові програмувані достатньо легко, так як в ньому мало допоміжних елементів, а правила мови заставляють програмістів робити відступи.

Отже, Python:

- Простий у використанні
- Легкий в засвоєнні
- Вільний і відкритий
- Мова високого рівня
- Портативність
- Інтерпретація
- Розширена бібліотека

Python – проста і мінімалістична мова. Читання хорошої програми на мові Python дуже нагадує читання англійського тексту, хоча і досить суворого. Така псевдо-кодова природа Python є однією з його найсильніших сторін. Вона дозволяє зосередитися на вирішенні завдання, а не на самій мові.

Python – це приклад вільного і відкритого програмного забезпечення – FLOSS (Free/Libré and Open Source Software). Простіше кажучи, можна вільно поширювати копії цього програмного забезпечення, читати його вихідні тексти, вносити зміни, а також використовувати його частини в програмах. В основі вільного ПЗ лежить ідея спільноти, яка ділиться своїми знаннями. Це одна з причин, за якими Python такий хороший: він був створений і постійно поліпшується співтовариством, яке просто хоче зробити його кращим.

При написанні програми на Python ніколи не доведеться відволікатися на такі низькорівневі деталі, як управління пам’яттю, використовуваної програмою.

Завдяки своїй відкритій природі, Python був імпортований на багато платформ (тобто змінений таким чином, щоб працювати на них). Всі програми зможуть запускатися на цих платформах без будь яких змін, якщо тільки уникати використання системно-залежних функцій.

Python можна використовувати в GNU/Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Аміга, AROS, AS/400, BeOS, OS / 390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS , VxWorks, PlayStation, Sharp Zaurus, Windows CE і навіть на PocketPC.

Програма, написана на компільованій мові програмування, як наприклад, C або C++, перетворюється з вихідної мови (тобто C або C ++) в мову, зрозумілу комп'ютеру (бінарний код, тобто нулі і одиниці) за допомогою компілятора із застосуванням різноманітних прапорів і параметрів. Коли запускається така програма, компонувальник/завантажувач копіює її з диска в оперативну пам'ять і запускає її.

Python не вимагає компіляції в бінарний код. Програма просто виконується з початкового тексту. Python сам перетворює цей вихідний текст в деяку проміжну форму, звану байткод, а потім переводить його на машинну мову і запускає. Все це помітно полегшує використання Python, оскільки немає необхідності клопотатись про компіляції програми, підключенні і завантаженні потрібних бібліотек і т.д. Разом з тим, це робить програми на Python набагато більш переносними, так як досить їх просто скопіювати на інший комп'ютер і вони працюють.

Мова програмування Python підтримує як процедурно-орієнтоване, так і об'єктно-орієнтоване програмування. У процедурно-орієнтованих мовах програми будуються на основі процедур або функцій, які представляють собою багаторазово використовувані фрагменти програми. В об'єктно-орієнтованих мовах програмування програми будуються на основі об'єктів, які об'єднують в собі дані і функціонал. Python надає прості але потужні засоби для ООП, особливо в порівнянні з такими великими мовами програмування, як C ++ або Java.

Стандартна бібліотека Python просто величезна. Вона може допомогти у вирішенні найрізноманітніших завдань, пов'язаних з використанням регулярних

виразів, генеруванням документації, перевіркою блоків коду, розпаралелюванням процесів, базами даних, веб-браузерами, CGI, FTP, електронною поштою, XML, XML-RPC, HTML, BAB файлами, криптографією, GUI (графічний інтерфейс користувача) та іншими системно-залежними речами. Пам’ятайте, що все це є на кожному кроці, де встановлений Python. У цьому полягає філософія Python «Все включено». Крім стандартної бібліотеки, існує безліч інших високоякісних бібліотек, які можна знайти в каталозі пакетів Python.

РОЗДІЛ 2. ЗАПУСК ПРОГРАМ НА PYTHON

Існує два способи запуску програм на Python: використання інтерактивного запрошення інтерпретатора і використання файлу з текстом програми.

2.1. Використання командного рядка інтерпретатора

Відкрити вікно терміналу і запусти інтерпретатор Python, ввівши команду python3 і натиснути Enter.

Користувачі Windows можуть запустити інтерпретатор в командному рядку, якщо встановили змінну PATH належним чином. Щоб відкрити командний рядок в Windows, потрібно зайти в меню «Пуск» і натиснути «Виконати ...». У діалоговому вікні ввести «cmd» і натиснути Enter; тепер буде все необхідне для початку роботи з Python в командному рядку DOS.

Якщо використовується IDLE, то Пуск → Програми → Python 3.0 → IDLE (Python GUI).

Як тільки запущено python3, повинно бути >>> на початку рядка, де можна набирати код. Це і називається командним рядком інтерпретатора.

Тепер ввести `print ('Hello World')` і натиснути клавішу Enter. В результаті повинні з'явитися слова "`Hello World`".

Ось приклад того, що можна побачити на екрані, якщо використовувати комп'ютер з Mac OS X. Інформація про версії Python може відрізнятися в залежності від комп'ютера, але частина, що починається із запрошення (тобто від >>> і далі) повинна бути однаковою на всіх операційних системах.

```
$ python3
Python 3.3.0 (default, Oct 22 2012, 12:20:36)
[GCC 4.2.1 Compatible Apple Clang 4.0 ((tags/Apple/clang-
421.0.60))] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> print ('Hello world')
hello world
>>>
```

Python видає результат роботи рядки миттєво. Одиночний «оператор» Python `print` використовується для того, щоб надрукувати будь яке передане в нього значення. В даному випадку – текст `"Hello World"`.

Для швидкого отримання інформації будь якої функції або оператора Python служить вбудована функція `help`. Це особливо зручно при використанні командного рядка інтерпретатора. Наприклад `help (print)` покаже довідку по функції `print`, яка використовується для виведення на екран.

2.2. Вибір редактора

Не можливо набирати програму в командному рядку інтерпретатора кожен раз, коли потрібно щось запустити. Тому знадобиться зберігати програми в файлах, щоб мати можливість запускати їх скільки завгодно разів.

Перш ніж приступити до написання програм на Python, потрібен редактор для роботи з файлами програм. Вибір редактора вкрай важливий. Підходити до вибору редактора слід так само, як і до вибору особистого автомобіля. Хороший редактор допоможе легко писати програми на Python, роблячи програмування більш комфортним, а також дозволяючи швидше і безпечніше досягти мети.

Одна з головних вимог – це підсвічування синтаксису, коли різні елементи програми на Python розфарбовані так, щоб можна легко бачити програму і хід її виконання.

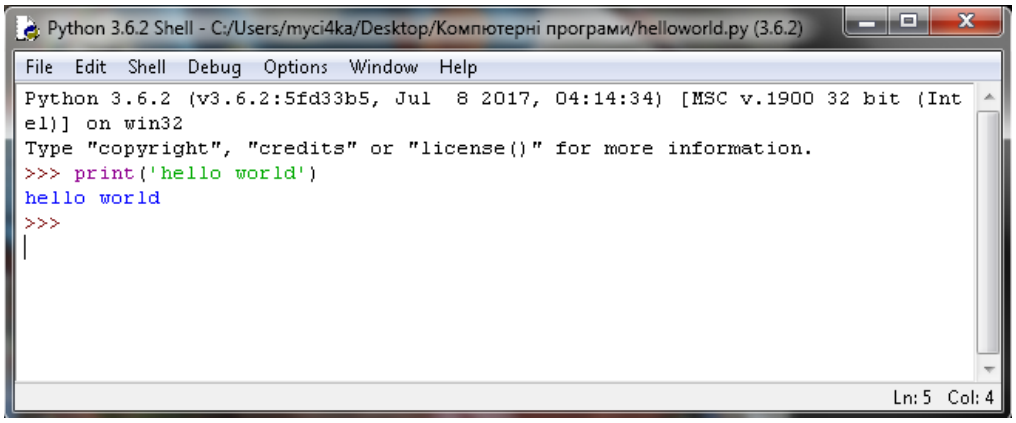
Якщо користуєтеся Windows, не варто використовувати Блокнот – це поганий вибір, оскільки він не має функції підсвічування синтаксису, а також не дозволяє автоматично вставляти відступи, що є дуже важливо. Хороші редактори, як Komodo Edit, дозволяють робити це автоматично. Найбільш потужні редактори: Vim та Emacs.

2.3. Використання програмних файлів

Існує така традиція, що якою б мовою програмування не починати вчити, першою програмою повинна бути програма «Привіт, світ!». Це програма, яка просто виводить напис `"Привіт, світ!"`.

Запустити обраний редактор, ввести наступну програму і зберегти її під ім’ям `helloworld.py`.

```
>>> print ('hello world')
```



Натиснути: Файл → Зберегти, для збереження файлу.

Створіть нову папку і використовуйте її для всіх ваших програм на Python:

`C:\\py в Windows`

РОЗДІЛ 3. ОСНОВИ PYTHON

Просто надрукувати "Привіт, світ!". Недостатньо, вірно? Потрібно зробити більше – ввести щось в програму, обробити і отримати щось на виході. В Python це можна організувати за допомогою констант і змінних, а також деякими іншими способами.

• Коментарі – це те, що пишеться після символу #, і представляє інтерес лише як замітка для читання програми.

```
Наприклад:
print ('Привіт, світ!')
# print – це функція
або:
# Print – це функція
print ('Привіт, світ!')
```

Краще в програмах писати якомога більше корисних коментарів, що пояснюють: припущення; важливі рішення; важливі деталі; проблеми, які намагаєтеся вирішити; проблеми, яких намагаєтеся уникнути і т.д.

Текст програми говорить про те, ЯК, а коментарі повинні пояснювати, ЧОМУ.

• Літеральні константи. Прикладом літеральної константи може бути число, наприклад, 5, 1.23, 9.25E-3 або що-небудь на зразок 'Це рядок' або "It's a string!". Вони називаються літеральними, тому що вони «буквальні» – використовуються їх значення буквально. Число 2 завжди представляє саме себе і нічого іншого – це «константа», тому що її значення неможна змінити. Тому все це називається літеральними константами.

• Числа. Числа в Python бувають трьох типів: цілі, з плаваючою крапкою і комплексні.

- Прикладом цілого числа може служити 2.

- Прикладами чисел з плаваючою точкою (або «плаваючих» для стислості) можуть бути 3.23 і 52.3E-4. Позначення E показує ступеня числа 10. В даному випадку 52.3E-4 означає $52.3 \cdot 10^{-4}$.

- Приклади комплексних чисел: (-5 + 4j) і (2.3 - 4.6j).

• Рядки – це послідовність символів. Найчастіше рядки – це просто деякі набори слів. Слова можуть бути як англійською мовою, так і на будь-якій іншій, підтримуваному стандартом Unicode.

- Одинарні лапки. Рядок можна вказати, використовуючи одинарні лапки, як наприклад, `'Фраза в кавичках'`. Всі прогалини і знаки табуляції збережуться.

- Подвійні лапки. Рядки в подвійних лапках працюють точно так же, як і в одинарних. Наприклад, `"What's your name?"`.

- Потрійні лапки. Можна вказувати «багаторядкові» рядки з використанням потрійних лапок (`""" """` або `''' '''`). В межах потрійних лапок можна вільно використовувати подвійні або потрійні лапки.

```
Наприклад:
'''Це багато строковий рядок. Це її перший рядок.
Це її другий рядок.
"What's your name?", - запитав я.
Він відповів: "Bond, James Bond."
'''
```

Рядки є незмінними – це означає, що після створення рядка його більше не можна змінювати. На перший погляд це може здатися недоліком, але насправді це не так.

3.1. Об’єднання строкових констант

Якщо розташувати поруч дві строкові константи, Python автоматично їх об’єднає. Наприклад, `'What \' s '' your name?'` автоматично перетворюється в `"What's your name?"`.

Іноді буває потрібно скласти рядок на основі будь яких даних. Тоді використовується метод `format ()`.

```
Зберегти наступні рядки в файл str_format.py:
age = 26
name = 'Swaroop'
print('Вік {0} -- {1} років.'.format(name, age))
print('Чому {0} забавляється з цим Python?'.format(name))

Виведення:
$ python str_format.py
```

```
Вік Swaroop -- 26 років.
Чому Swaroop забавляється з цим Python?
```

3.2. Змінні

Використання одних лише літеральних констант може швидко стати не актуальним, оскільки потрібно зберігання інформації та маніпулювання нею. Ось тут на сцену виходять змінні. Слово «змінні» говорить саме за себе – їх значення може змінюватися, а значить можна зберігати в змінній все, що завгодно. Змінні – це просто область пам'яті комп'ютера, в яких зберігається деяка інформація. На відміну від констант, до такої інформації потрібно якимось чином отримувати доступ, тому змінним даються імена.

Змінні – це окремий випадок ідентифікаторів. Ідентифікатори – це імена, присвоєння чогось для його позначення. При виборі імен для ідентифікаторів необхідно дотримуватися таких правил:

• Першим символом ідентифікатора повинна бути буква з алфавіту (символ ASCII в верхньому або нижньому регістрі, або символ Unicode), а також символ підкреслення ("_").

• Інша частина ідентифікатора може складатися з букв (символи ASCII в верхньому або нижньому регістрі, а також символи Unicode), знаків підкреслення ("_") або цифр (0-9).

• Імена ідентифікаторів чутливі до регістру. Наприклад, `myname i myName` – це не одне і те ж. Варто звернути увагу на `"n"` у нижньому регістрі в першому випадку і `"N"` у верхньому в другому.

• Приклади допустимих імен ідентифікаторів: `i`, `__my_name`, `name_23`, `a1b2_c3` і будь `які_символи_utf8_δξѪђёўЩлΞєά`.

• Приклади неприпустимих імен ідентифікаторів: `2things`, `тут є прогалини`, `my-name`, `> a1b2_c3` і `"це_в_кавичках"`.

Змінні можуть зберігати значення різних типів, званих типами даних. Основними типами є числа і рядки. В подальших розділах побачимо, як створювати власні типи за допомогою класів.

Python розглядає все, що є в програмі, як об'єкти. Мається на увазі, в найзагальнішому сенсі. Замість того, щоб говорити "щось", говоримо "об'єкт".

Як писати програми на Python. Надалі стандартна процедура збереження і запуску програми на Python буде виглядати так:

1. Відкрити улюблений редактор.

2. Ввести текст програми з прикладу.

3. Зберегти його в файл, вказавши його ім'я в коментарі. Потрібно дотримуватись правило зберігати всі програми на Python з розширенням `.py`.

4. Запустити інтерпретатор командою python3 `program.py`.

```
Приклад: Використання змінних і констант
# Ім'я файлу : var.py
i = 5
print (i)
i = i + 1
print (i)
s = '''Це багато строковий рядок.
Це другий її рядок.'''
print (s)

Виведення:
$ python var.py
5
6
Це багато строковий рядок.
Це другий її рядок.
```

Як це працює. Спершу присвоюємо значення константи `5` змінній `i`, використовуючи оператор присвоювання (=). Цей рядок називається пропозицією і вказує, що має бути зроблена дія. В даному випадку пов'язуємо ім'я змінної `i` зі значенням `5`. Потім виводимо значення `i`, використовуючи функцію `print`, яка просто виводить значення змінної на екрані.

Далі додаємо `1` до значення, що зберігається в `i` та зберігаємо його там. Після цього виводимо його і отримуємо значення `6`.

Аналогічним чином присвоюємо строкову константу змінної `s`, після чого виводимо її.

3.3. Логічні та фізичні рядки

Фізичний рядок – це те, що бачимо, коли набираємо програму. Логічний рядок – це те, що Python бачить як єдину пропозицію. Python неявно передбачає, що кожному фізичному рядку відповідає логічний рядок.

Прикладом логічного рядка може служити вираз `print ('Привіт, світ!')` – якщо воно на одному рядку, то цей рядок також відповідає фізичному рядку.

Python неявно стимулює використання по одному реченню на рядок, що полегшує читання коду. Щоб записати більше одного логічного рядка на одному фізичному рядку, доведеться явно вказати це за допомогою крапки з комою (`;`), яка відзначає кінець логічного рядка/пропозиції.

```
Наприклад:
i = 5
print (i)
те саме, що
i = 5;
print (i);
і те ж саме може бути записано у вигляді
i = 5; print (i);
або
i = 5; print (i)
```

Однак варто дотримуватися написання одного логічного рядка в кожному фізичному рядку. Таким чином можна обійтися зовсім без крапки з комою.

В Python є важливими пробіли. Точніше, пробіли на початку рядка. Це називається відступами. Передні відступи (пробіли і табуляції) на початку логічного рядка використовуються для визначення рівня відступу логічного рядка, який, в свою чергу, використовується для угруповання пропозицій.

Це означає, що пропозиції, які йдуть разом, повинні мати однаковий відступ. Кожен такий набір пропозицій називається блоком. У подальших розділах побачимо приклади того, наскільки важливі блоки.

Варто запам’ятати, що неправильні відступи можуть призводити до виникнення помилок.

```
Наприклад:
i = 5
```

```
      print('Значення відповідає ', i) # Помилка! Пробіл на початку
рядка
      print('Я повторюю, значення відповідає ', i)
     Коли запустите це, отримаєте наступну помилку:
     File "whitespace.py", line 4
     print(''Значення відповідає'', i) # Помилка! Пробіл на початку
рядка
     ^
     IndentationError: unexpected indent
```

На початку другого рядка є один пробіл. Помилка, відображена Python, говорить про те, що синтаксис програми є неправильним, тобто програма не була написана за правилами. Це означає, що не можна починати нові блоки пропозицій де попало (крім основного блоку за замовчуванням, який використовується на протязі всієї програми, звичайно).

РОЗДІЛ 4. ОПЕРАТОРИ І ВИРАЗИ

Більшість пропозицій (логічних рядків) в програмах містять вирази. Простий приклад виразу: `2 + 3`. Вираз можна розділити на оператори і операнди.

Оператори – це функціонал, що виконує будь які дії, які можуть бути представлені у вигляді символів, як наприклад +, або спеціальних зарезервованих слів. Оператори можуть виконувати дії над даними, і ці дані називаються операндами. У нашому випадку `2` і `3` – це операнди.

Коротко розглянемо оператори та їх застосування. Обчислити значення виразів, даних в прикладах, можна також використовуючи інтерпретатор інтерактивно. Наприклад, для перевірки вираження `2+3` скористатись інтерактивним командним рядком інтерпретатора Python:

```
>>> 2 + 3
5
>>> 3 * 5
15
```

Оператори і їх застосування (додаток 1)

Найчастіше результатам проведення якоїсь математичної операції необхідно присвоїти змінну, над якою ця операція проводилася. Для цього існують короткі форми запису виразів:

Можна записати:

```
a = 2; a = a * 3
```

у вигляді

```
a = 2; a *= 3
```

Варто звернути увагу, що вирази виду «`змінна = змінна операція вираз`» набирає вигляду «`змінна операція = вираз`».

Якщо є вираз вигляду `2+3*4`, що виконується раніше: додавання чи множення? Шкільний курс математики говорить, що множення має проводитися в першу чергу. Це означає, що оператор множення має вищий пріоритет, ніж оператор додавання.

Наступна таблиця показує пріоритет операторів в Python, починаючи з найнижчого (найслабкіше зв’язування) і до найвищого (найсильніше зв’язування). Це означає, що в будь якому вираженні Python спершу обчислює оператори і вирази, розташовані внизу таблиці, а потім оператори вище по таблиці.

Таблиця 1

Пріоритети операторів

Оператор	Опис
lambda	Лямбда-вираз
or	Логічне “або”
and	Логічне “і”
not x	Логічне “не”
in, not in	Перевірка приналежності
is, is not	Перевірка тотожності
<, <=, >, >=, !=, ==	Порівняння
|	Побітове “або”
^	Побітове “виключно або”
&	Побітове “і”
<<, >>	Зміщення
+, -	Додавання і віднімання
*, /, //, %	Множення, ділення, цілочисельне ділення і залишок
+x, -x	Позитивне, негативне
~x	Побітове “не”
**	Зведення в ступінь
x.attribute	Посилання на атрибут
x[індекс]	Звернення за індексом
x[індекс1:індекс2]	Вирізка
f(аргумент ...)	Виклик функції
(вираз, ...)	Зв'язка або кортеж
[вираз, ...]	Список
{ключ:дані, ...}	Словник

Для полегшення читання виразів можна використовувати дужки. Наприклад, `2+(3*4)` легше зрозуміти, ніж `2+3*4`, яке вимагає знання пріоритету операторів. Дужки потрібно використовувати розумно, тобто уникати зайвих `(2 + (3 * 4))`.

Існує ще одна перевага у використанні дужок – вони дають можливість змінити порядок обчислення виразів. Наприклад, якщо складання необхідно провести перш множення, можна записати щось на зразок `(2 + 3) * 4`.

Оператори зазвичай обробляються зліва направо. Це означає, що оператори з рівним пріоритетом будуть оброблені по порядку від лівого до правого. Наприклад,

2+3+4 обробляється як (2+3)+4. Деякі оператори, як наприклад, оператор присвоювання, мають асоціативність справа наліво, тобто a = b = c розглядається як a = (b = c).

```
Приклад (зберегти як expression.py):
length = 5
breadth = 2
area = length * breadth
print ('Площа рівна', area)
print ('Периметр рівний', 2 * (length + breadth))

Виведення
$ python expression.py
Площа рівна 10
Периметр рівний 14
```

Як це працює. Довжина і ширина прямокутника зберігаються в змінних length і breadth відповідно. Використовуємо їх для обчислення периметра і площі прямокутника за допомогою виразів. Результат вираження length * breadth зберігається в змінної area, після чого виводиться на екран функцією print. У другому випадку безпосередньо підставляємо значення виразу 2 * (length + breadth) в функцію print.

Варто звернути увагу, як Python «красиво друкує» результат. Незважаючи на те, що не вказано пробіл між 'Площа дорівнює' і змінною area, Python підставляє його за нас, щоб отримати гарний і зрозумілий висновок. Програма ж залишається при цьому легко читаюча.

РОЗДІЛ 5. ПОТІК КОМАНД

Для того щоб програма приймала деяке рішення і виконувала різні дії в залежності від ситуації; скажімо, друкувала "Доброго ранку" або "Доброго вечора" в залежності від часу доби, існують оператори управління потоком. В Python є три оператора управління потоком: `if`, `for` і `while`.

Оператор `if`. Використовується для перевірки умов: якщо умова правильна, виконується блок виразів (званий «`if-блок`»), інакше виконується інший блок виразів (званий «`else-блок`»). Блок «`else`» є необов'язковим.

```
Приклад: (зберегти як if.py)
number = 23
guess = int (input ('Введіть ціле число : '))
if guess == number:
print ('Вітаю, ви вгадали,') # Тут починається новий блок
print ('(Хоча і не виграли жодного призу!)') # Тут завершується
новий блок
elif guess < number:
print ('Ні, загадане число трохи більше цього.') # Ще один блок
# В середині блоку ви можете виконувати будь які дії ...
else:
print ('Ні, загадане число трохи менше цього.')
# щоб перейти в необхідний діапазон, guess повинне бути більшим,
ніж number
print ('Завершено')
# Останній вираз виконується завжди після виконання оператора if

Виведення
$ Python if.py
Введіть ціле число : 50
Ні, загадане число трохи менше цього.
Завершено
$ Python if.py
Введіть ціле число : 22
Ні, загадане число трохи більше цього.
Завершено
$ Python if.py
Введіть ціле число : 23
Вітаю, ви вгадали,
(Хоча і не виграли жодного призу.)
Завершено
```

Як це працює. У цій програмі приймаємо варіанти від користувача і перевіряємо чи збігаються вони з наперед заданим числом. Встановлюємо змінній `number` значення будь якого цілого числа, якого хочемо. Наприклад, `23`. Після

цього приймаємо варіант числа від користувача за допомогою функції `input ()`. Функції – це всього-на-всього багаторазово використовувані фрагменти програми.

Передаємо вбудованій функції `input` рядок, яку вона виводить на екран і очікує введення від користувача. Як тільки ввели що-небудь і натиснули клавішу Enter, функція `input ()` повертає рядок, який ввели. Потім перетворюємо отриманий рядок в число за допомогою `int ()`, і зберігаємо це значення в змінну `guess`. Взагалі-то, `int` – це клас, але на даному етапі досить знати лише, що за допомогою нього можна перетворити рядок в ціле число (припускаючи, що рядок містить ціле число).

Далі порівнюємо число, введене користувачем, з числом, яке вибрано заздалегідь. Якщо вони рівні, друкуємо повідомлення про успіх. Зверніть увагу, що використовується відповідні рівні відступу, щоб вказати Python, які вислови відносяться до якогось блоку. Ось чому відступи так важливі в Python.

В кінці оператора `if` стоїть двокрапка – цим показано, що далі йде блок виразів. Після цього перевіряємо, чи правильно, що призначений для користувача варіант числа менше задуманого, і якщо це так, інформуємо користувача про те, що йому слід вибирати числа трохи більше цього. Тут використано вираз `elif`, який поєднує в собі два пов'язаних `if else-if else` вираження в один вираз `if-elif-else`. Це полегшує читання програми, а також не вимагає додаткових відступів.

Вирази `elif` і `else` також мають двокрапку в кінці логічного рядка, за яким слідують відповідні блоки команд (з відповідним числом відступів).

Усередині `if-блоку` оператора `if` може бути інший оператор `if` і так далі – це називається вкладеним оператором `if`.

Частини `elif` і `else` не обов'язкові. Мінімально коректний запис оператора `if`:

```
if True:
print ('Так, це вірно.')
```

Після того, як Python закінчує виконання всього оператора `if` разом з його частинами `elif` і `else`, він переходить до наступного виразу в блоці, що містить

цей оператор `if`. У нашому випадку це основний блок програми (в якому починається виконання програми), а такий вираз – це `print ('Завершено')`. Після цього Python доходить до кінця програми і просто виходить з неї.

Оператор `while`. Він дозволяє багаторазово виконувати блок команд до тих пір, поки виконується деяка умова. Це один з так званих операторів циклу. Він також може мати необов'язковий вираз `else`.

```
Приклад: (зберегти як while.py)
number = 23
running = True
hile running:
guess = int (input ('Введіть ціле число :'))
if guess == number :
print ('Вітаю, ви вгадали.')
running = False # це зупиняє цикл while
elif guess < number :
print ('Ні, загадане число трохи більше цього')
else:
print ('Ні, загадане число трохи менше цього.')
else:
print ('Цикл while закінчений.')
# Тут можете виконати все що вам ще потрібно
print ('Завершення.')

Виведення:
$ Python while.py
Введіть ціле число : 50
Ні, число дещо менше.
Введіть ціле число : 22
Ні, число дещо більше.
Введіть ціле число : 23
Вітаю, ви вгадали.
Цикл while закінчений.
Завершення.
```

Як це працює. У цій програмі продовжуємо «грати у гру вгадування», але перевага полягає в тому, що тепер користувач може вгадувати до тих пір, поки не вгадає правильне число, і йому не доведеться запускати програму заново для кожної спроби, як це відбувалося до цих пір. Це наочно демонструє застосування оператора `while`.

Оператори `input` і `if` переміщено всередину циклу `while` і встановлено змінну `running` в значення `True` перед запуском циклу. Перш за все

перевіряється, чи рівне значення змінної running True, а потім відбувається перехід до відповідного while-блоку. Після виконання цього блоку команд умова, яка в даному випадку є змінна running, перевіряється знову. Якщо воно істинне, while-блок запускається знову, в іншому випадку відбувається перехід до додаткового else-блоку, а потім – до наступного оператору.

Блок else виконується тоді, коли умова циклу while стає хибним (False) – це може статися навіть при найпершій перевірці умови. Якщо у циклу while є додатковий блок else, він завжди виконується, якщо тільки цикл не буде перерваний оператором break.

True і False називаються логічним типом даних, і можна вважати їх еквівалентними значеннями 1 і 0 відповідно.

Цикл for. Оператор for..in також є оператором циклу, який здійснює ітерацію по послідовності об'єктів, тобто проходить через кожен елемент в послідовності. Послідовність – це упорядкований набір елементів.

```
Приклад: (збережіть як for.py)
for i in range (1, 5):
print (i)
else:
print ('Цикл for закінчений')

Виведення:
$ Python for.py
1
2
3
4
Цикл for закінчений
```

Як це працює. У цій програмі виводимо на екран послідовність чисел. Генеруємо цю послідовність, використовуючи вбудовану функцію range. Задаємо два числа, і range повертає послідовність чисел від першого числа до другого. Наприклад, range (1, 5) дає послідовність [1, 2, 3, 4]. За замовчуванням range приймає значення кроку, рівне 1. Якщо ми задамо також і третє число

range, воно буде служити кроком. Наприклад, range (1, 5, 2) дасть [1, 3]. Інтервал простягається тільки до другого числа, тобто не включає його в себе.

Варто звернути увагу, що range () генерує послідовність чисел, але тільки по одному числу за раз – коли оператор for запитує наступний елемент. Щоб побачити всю послідовність чисел відразу, використовується list (range ()).

Потім цикл for здійснює ітерацію з цього діапазону – for і in range (1, 5) еквівалентно for i in [1, 2, 3, 4], що нагадує присвоювання змінної і по одному числу (або об'єкту) за раз, виконуючи блок команд для кожного значення i. В даному випадку в блоці команд просто виводимо значення на екран.

Блок else не обов'язковий. Якщо він присутній, то завжди виконується один раз після закінчення циклу for, якщо тільки не вказано оператор break.

Цикл for..in працює для будь-якої послідовності. У нашому випадку це список чисел, згенерований вбудованою функцією range, але в загальному випадку можна використовувати будь-яку послідовність будь-яких об'єктів.

Оператор break. Оператор break служить для переривання циклу, тобто зупинення виконання команд навіть якщо умова виконання циклу ще не прийняла значення False або послідовність елементів не закінчилася.

Важливо відзначити, що якщо цикли for або while перервати оператором break, відповідні їм блоки else виконуватися не будуть.

```
Приклад: (зберегти як break.py)
while True:
s = input ('Введіть що-небудь :')
if s == 'вихід' :
break
print ('Довжина рядка :', len (s))
print ('Завершення')

Виведення
$ Python break.py
Введіть що-небудь : Програмувати весело.
Довжина рядка : 23
Введіть що-небудь : Якщо робота нудна,
Довжина рядка : 19
Введіть що-небудь : Щоб надати їй веселий тон
Довжина рядка : 30
Введіть що-небудь : використовуй Python!
```

```
Довжина рядка : 23
Введіть що-небудь : вихід
Завершення
```

Як це працює. У цій програмі багато разів зчитуємо користувача введення і виводимо на екран довжину кожного введеного рядка. Для зупинки програми вводимо спеціальну умову, що перевіряє, чи збігається призначений для користувача введення з рядком `'вихід'`. Зупиняємо програму перериванням циклу оператором `break` і досягаємо її кінця.

Довжина введеного рядка може бути знайдена за допомогою вбудованої функції `len`, що оператор `break` може застосовуватися і в циклі `for`.

Оператор `continue`. Використовується для вказівки Python, що необхідно пропустити всі команди, які наразі в поточному блоці циклу і продовжити з наступною ітерацією циклу.

```
Приклад: (зберегти як continue.py)
while True :
s = input ('Введіть що-небудь :')
if s == 'вихід':
break
if len (s) < 3 :
print ('Занадто мало')
continue
print ('Введено рядок достатньої довжини')
# Різні інші дії тут ...

Виведення
$ Python continue.py
Введіть що-небудь : а
Замало
Введіть що-небудь : 12
Замало
Введіть що-небудь : абв
Введено рядок достатньої довжини
Введіть що-небудь : вихід
```

Як це працює. Використовується вбудована функція `len` для отримання довжини рядка, і якщо довжина менше `3`, пропускаємо інші дії в блоці за допомогою оператора `continue`. В іншому випадку всі інші команди в циклі виконуються, виробляючи будь-які маніпуляції, які потрібні.

Оператор `continue` також працює і з циклом `for`.

РОЗДІЛ 6. ФУНКЦІЇ

Функції – це багаторазово використовувані фрагменти програми. Вони дозволяють дати ім’я певного блоку команд з тим, щоб надалі запускати цей блок по зазначеному імені в будь-якому місці програми і скільки завгодно разів. Це називається викликом функції.

Функції визначаються за допомогою зарезервованого слова `def`. Після цього слова вказується ім’я функції, за яким слідує пара дужок, в яких можна вказати імена деяких змінних, і заключне двокрапка в кінці рядка. Далі йде блок команд, складових функції. На прикладі можна бачити, що насправді це дуже просто:

```
Приклад: (зберегти як function1.py)
def sayHello () :
print ('Привіт, світ!') # блок, що належить функції
# Кінець функції
sayHello () # виклик функції
sayHello () # ще один виклик функції

Виведення:
$ Python function1.py
Привіт світ!
Привіт світ!
```

Як це працює. Визначено функцію з ім’ям `sayHello`, використовуючи описаний вище синтаксис. Ця функція не приймає параметрів, тому в дужках не показані будь які змінні. Параметри функції – це такі собі вхідні дані, які можна передати функції, щоб отримати відповідний їм результат.

Одну й ту ж функцію можна викликати багато разів, а значить немає необхідності писати один і той же код знову і знову.

Функції можуть приймати параметри, тобто деякі значення, що передаються функції для того, щоб вона щось зробила з ними. Ці параметри схожі на змінні, за винятком того, що значення цих змінних вказується при виклику функції, і під час роботи функції їм уже присвоєні значення.

Параметри вказуються в дужках при оголошенні функції і розділяються комами. Аналогічно передаємо значення, коли викликаємо функцію. Термінологія:

імена, зазначені в оголошенні функції, називаються параметрами, тоді як значення, які передаються в функцію при її виклику – аргументами.

```
Приклад: (зберегти як func_param.py)
def printMax (a, b):
if a> b :
print (a, 'максимально')
elif a == b :
print (a, 'дорівнює', b)
else:
print (b, 'максимально')
printMax (3, 4) # пряма передача значень
x = 5
y = 7
printMax (x, y) # передача змінних в якості аргументів

Виведення
$ python func_param.py
4 максимально
7 максимально
```

Як це працює. Визначено функцію з ім'ям `printMax`, яка використовує два параметра з іменами `a` і `b`. Знаходимо найбільше число із застосуванням простого оператора `if..else` і виводимо це число.

При першому виклику функції `printMax` безпосередньо передаємо числа в якості аргументів. У другому випадку викликаємо функцію зі змінними в якості аргументів. `PrintMax (x, y)` призначає значення аргументу `x` параметру `a`, значення аргументу `y` – параметру `b`. В обох випадках функція `printMax` працює однаково.

Локальні змінні. При представлені змінних всередині визначення функції, вони жодним чином не пов'язані з іншими змінними з таким же ім'ям за межами функції – тобто імена змінних є локальними в функції. Це називається областю видимості змінної. Область видимості всіх змінних обмежена блоком, в якому вони представлені, починаючи з точки виведення імені.

```
Приклад: (збережіть як func_local.py)
x = 50
def func (x) :
print ('x дорівнює', x)
x = 2
print ('Заміна локального x на', x)
func (x)
```

```
print ('x і раніше', x)

Виведення
$ Python func_local.py
x дорівнює 50
Заміна локального x на 2
x і раніше 50
```

Як це працює. При першому виведенні значення, присвоєного імені `x`, в першому рядку функції Python використовує значення параметра, показаного в основному блоці, вище визначення функції.

Далі призначаємо `x` значення `2`. Ім'я `x` локальне для функції. При цьому коли замінюємо значення `x` в функції, показаний в основному блоці, залишається незмінним.

Останнім викликом функції `print` виводимо значення `x`, вказане в основному блоці, підтверджуючи таким чином, що воно не змінилося при локальному присвоєнні значення в раніше викликаної функції.

Зарезервоване слово «`global`». Щоб привласнити деяке значення змінній, визначеній на вищому рівні програми, необхідно вказати Python, що її ім'я не локальне, а глобальне. Це можна зробити за допомогою зарезервованого слова `global`. Без застосування зарезервованого слова `global` неможливо привласнити значення змінної, визначеної за межами функції.

Можна використовувати вже існуючі значення змінних, визначених за межами функції (за умови, що всередині функції не було показано змінної з таким же ім'ям). Однак, це не вітається, і його слід уникати, оскільки людині, що читає текст програми, буде незрозуміло, де знаходиться вказана змінна. Використання зарезервованого слова `global` досить ясно показує, що змінна представлена в самому зовнішньому блоці.

```
Приклад: (збережіть як func_global.py)
x = 50
def func ():
global x
print ('x рівно', x)
x = 2
print ('Замінюємо глобальне значення x на', x)
func ()
```

```
print ('Значення x становить', x)

Виведення
$ Python func_global.py
x дорівнює 50
Замінюємо глобальне значення x на 2
Значення x становить 2
```

Як це працює. Зарезервоване слово `global` використовується для того, щоб оголосити, що `x` – це глобальна змінна, а значить, коли присвоюємо значення імені `x` всередині функції, це зміна відіб'ється на значенні змінної `x` в основному блоці програми.

Використовуючи одне зарезервоване слово `global`, можна оголосити відразу кілька змінних: `global x, y, z`.

Зарезервоване слово «`nonlocal`». Є ще один тип області видимості, званна «нелокальної» (`nonlocal`) область видимості, яка є чимось середнім між першими двома. Нелокальні області видимості зустрічаються, коли визначається функції всередині функцій.

```
Приклад:
# Filename: func_nonlocal.py
def func_outer () :
x = 2
print ('x рівне', x)
def func_inner () :
nonlocal x
x = 5
func_inner ()
print ('Локальне x змінилося на', x)
func_outer ()

Виведення
$ Python func_nonlocal.py
x дорівнює 2
Локальне x змінилося на 5
```

Як це працює. Коли знаходимося всередині `func_inner`, змінна `x`, визначена в першому рядку `func_outer` знаходиться ні в локальній області видимості (визначення змінної не входить в блок `func_inner`), ні в глобальному контексті (вона також і не в основному блоці програми). Показуємо, що хочемо використовувати саме цю змінну `x`, в такий спосіб: `nonlocal x`.

Замінимо «nonlocal x» на «global x», а потім видалити це зарезервоване слово, і спостерігаємо за різницею між цими двома випадками.

6.1. Значення аргументів

Аргумент за замовчуванням. Найчастіше частина параметрів функцій можуть бути необов'язковими, і для них будуть використовуватися деякі задані значення за замовчуванням, якщо користувач не вкаже власних. Цього можна досягти за допомогою значень аргументів за замовчуванням. Їх можна вказати, додавши до імені параметра у визначенні функції оператор присвоювання (=) з подальшим значенням.

Значення за замовчуванням має бути константою або точніше кажучи, воно має бути незмінним.

```
Приклад: (зберегти як func_default.py)
def say (message, times = 1) :
print (message * times)
say ('Привіт')
say ('Світ', 5)

Виведення
$ Python func_default.py
Привіт
СвітСвітСвітСвітСвіт
```

Як це працює. Функція під ім'ям say використовується для виведення на екран рядків вказане число раз. Якщо не вказуємо значення, за умовчанням рядок виводиться один раз. Досягнувши цього значення аргументу за замовчуванням, рівного 1 для параметра times.

При першому виклику say вказуємо тільки рядок, і функція виводить її один раз. При другому виклик say вказуємо також і аргумент 5, позначаючи таким чином, що хочемо сказати вислів 5 разів.

Ключові аргументи. Якщо є деяка функція з великим числом параметрів, і при її виклику необхідно вказати тільки деякі з них, значення цих параметрів можуть задаватися по їх імені – це називається ключовими параметрами. В цьому

випадку для передачі аргументів функції використовується ім’я (ключ) замість позиції (як було досі).

Є дві переваги такого підходу: по-перше, використання функції стає легше, оскільки немає необхідності відстежувати порядок аргументів; по-друге, можна задавати значення тільки деяким обраним аргументам, за умови, що решта параметрів мають значення аргументу за замовчуванням.

```
Приклад: (збережіть як func_key.py)
def func (a, b = 5, c = 10) :
print ('a рівне', a, ', b рівне', b, ', a c рівне', c)
func (3, 7)
func (25, c = 24)
func (c = 50, a = 100)

Виведення
$ Python func_key.py
a рівне 3, b рівне 7, a c рівне 10
a рівне 25, b рівне 5, a c рівне 24
a рівне 100, b рівне 5, a c рівне 50
```

Як це працює. Функція з ім’ям `func` має один параметр без значення по замовчуванню, за яким слідують два параметра із значеннями по замовчуванню.

При першому виклику `func (3, 7)` параметр `a` отримує значення `3`, параметр `b` отримує значення `7`, параметр `c` отримує своє значення за замовчуванням, рівне `10`.

При другому виклик `func (25, c = 24)` змінна `a` отримує значення `25` в силу позиції аргументу. Після цього параметр c отримує значення `24` по імені, тобто як ключовий параметр. Мінлива `b` отримує значення за замовчуванням, рівне `5`.

При третьому зверненні `func (c = 50, a = 100)` використовуємо ключові аргументи для всіх зазначених значень.

Вказується значення для параметра `c` перед значенням для `a`, навіть незважаючи на те, що у визначенні функції параметр `a` вказано раніше `c`.

6.2. Змінна числа параметрів

Для того, щоб визначити функцію, здатну приймати будь яке число параметрів, необхідно: (зберегти як total.py):

```
def total (initial = 5, * numbers, * * keywords):
count = initial
for number in numbers :
count += number
for key in keywords :
count += keywords[key]
return count

print (total(10, 1, 2, 3, vegetables=50, fruits=100))

Виведення
$ python total.py
166
```

Як це працює. Коли оголошуємо параметр з * (наприклад, * param), всі позиційні аргументи починаючи з цієї позиції і до кінця будуть зібрані в кортеж під ім'ям param.

Аналогічно, коли оголошуємо параметри з двома ** (** param), всі ключові аргументи починаючи з цієї позиції і до кінця будуть зібрані в словник під ім'ям param.

Якщо деякі ключові параметри повинні бути доступні тільки по ключу, а не як позиційні аргументи, їх можна оголосити після параметра із * (зберегти як keyword_only.py):

```
def total (initial=5, *numbers, extra_number):
count = initial
for number in numbers :
count += number
count += extra_number
print (count)
total (10, 1, 2, 3, extra_number=50)
total (10, 1, 2, 3)

# Викличе помилку, оскільки не вказали значення
# Аргументу за замовчуванням для 'extra_number'.

Виведення
$ python keyword_only.py
66
Traceback (most recent call last):
File "keyword_only.py", line 12, in <module>
total(10, 1, 2, 3)
TypeError: total() needs keyword-only argument extra_number
```

Як це працює. Оголошення параметрів після параметра із * дає лише ключові аргументи. Якщо для таких аргументів не вказано значення за замовчуванням, і воно не передано при виклику, звернення до функції викличе помилку.

Варто звернути увагу на використання + =, який представляє собою спрощенний оператор, що дозволяє замість x = x + y написати x + =y.

Якщо потрібні аргументи, що передаються тільки по ключу, але не потрібен параметр із *, то можна просто вказати одну зірочку без вказівки імені: def total (initial = 5, *, extra_number).

Оператор `return` використовується для повернення з функції, тобто для припинення її роботи і виходу з неї. При цьому можна також повернути деяке значення з функції.

```
Приклад: (зберегти як func_return.py)
#!/usr/bin/python
# Filename: func_return.py
def maximum(x, y) :
if x > y :
return x
elif x == y :
return 'Числа рівні.'
else:
return y
print (maximum (2, 3))

Виведення
$ python func_return.py
3
```

Як це працює. Функція maximum повертає максимальний з двох параметрів, які в даному випадку передаються їй при виклику. Вона використовує звичайний умовний оператор if..else для визначення найбільшого числа, а потім повертає це число.

Зверніть увагу, що оператор return без вказівки повертається еквівалентний висловом return None. None – це спеціальний тип даних в Python, що позначає нічого. Наприклад, якщо значення змінної встановлено в None, це означає, що їй не присвоєно ніякого значення.

6.3. Рядки документації

Python характерна особливість – рядки документації, які позначаються скорочено `docstrings`. Це дуже важливий інструмент, який допомагає краще документувати програму і полегшує її розуміння. Рядок документації можна отримати, наприклад, з функції, навіть під час виконання програми.

```
Приклад: (зберегти як func_doc.py)
def printMax (x, y):
'' 'Виводить максимальне з двох чисел.
Обидва значення повинні бути цілими числами. '' '
x = int (x) # конвертуємо в цілі, якщо можливо
y = int (y)
if x > y :
print (x, 'найбільше')
else:
print (y, 'найбільше')

printMax (3, 5)
print (printMax .__ doc__)

Виведення
$ Python func_doc.py
5 найбільше
Виводить максимальне з двох чисел.

Обидва значення повинні бути цілими числами.
```

Як це працює. Рядок в першому логічному рядку функції є рядком документації для цієї функції. Рядки документації прийнято записувати у формі багато строкових рядків, де перший рядок починається з великої літери і закінчується крапкою. Другий рядок залишається порожнім, а докладний опис починається з третьої.

Доступ до рядка документації функції `printMax` можна отримати за допомогою атрибута цієї функції (тобто імені, що належить їй) `__doc__` (зверніть увагу на подвійне підкреслення).

РОЗДІЛ 7. МОДУЛІ

Для повторного використання функцій в інших програмах застосовують модулі. Існують різні способи складання модулів, але найпростіший – це створити файл з розширенням `.py`, що містить функції і змінні.

Інший спосіб – написати модуль на тій мові програмування, на якому написаний сам інтерпретатор Python. Наприклад, можна писати модулі на мові програмування C, які після компіляції можуть використовуватися стандартним інтерпретатором Python.

Модуль можна імпортувати в іншу програму, щоб використовувати функції з нього. Точно так само використовуємо стандартну бібліотеку Python. Спершу подивимося, як використовувати модулі стандартної бібліотеки.

```
Приклад: (зберегти як using_sys.py)
import sys
print ('Аргументи командного рядка :')
for i in sys.argv :
print (i)
print ('\n\n Змінна PYTHONPATH містить', sys.path, '\ n'))

Виведення
$ Python3 using_sys.py we are arguments
Аргументи командного рядка:
using_sys.py
we
are
arguments

Змінна PYTHONPATH містить ['', 'C: \\ Windows \\ system32 \\
python30.zip',
'C: \\ Python30 \\ DLLs', 'C: \\ Python30 \\ lib',
'C: \\ Python30 \\ lib \\ plat-win', 'C: \\ Python30',
'C: \\ Python30 \\ lib \\ site-packages']
```

Як це працює. На початку імпортуємо модуль `sys` командою `import`. Цим говоримо Python, що хочемо використовувати цей модуль. Модуль `sys` містить функції, що відносяться до інтерпретатора Python і його середовищі, тобто до системи `(system)`.

Коли Python виконує команду `import sys`, він шукає модуль `sys`. В даному випадку це один з вбудованих модулів, і Python знає, де його шукати.

Якби це був не скомпільований модуль, тобто модуль, написаний на Python, тоді інтерпретатор Python шукав би його в каталогах, перерахованих у змінній `sys.path`. Якщо модуль знайдений, виконуються команди в тілі модуля, і він стає доступним. Ініціалізація відбувається тільки при початковому імпорті модуля.

Доступ до змінної `argv` в модулі `sys` надається за допомогою точки, тобто `sys.argv`. Це явно показує, що ім'я є частиною модуля `sys`. Ще однією перевагою такого позначення є те, що ім'я не конфліктує з ім'ям змінної `argv`, яка може використовуватися у програмі.

Змінна `sys.argv` – список рядків. Містить список аргументів командного рядка, тобто аргументів, переданих програмі.

Якщо використовується середовище розробки для написання і запуску програм, тоді в меню є можливість передавати параметри командного рядка.

У показаному прикладі, коли запускаємо "`python using_sys.py we are arguments`", запускаємо модуль `using_sys.py` командою `python`, а все, що слідує далі – аргументи, що передаються програмі. Python зберігає аргументи командного рядка в змінної `sys.argv` для подальшого використання.

Ім'я запускаючого скрипту завжди є першим аргументом в списку `sys.argv`. Так що в наведеному прикладі `'using_sys.py'` буде елементом `sys.argv [0]`, `'we' - sys.argv [1]`, `'are' - sys.argv [2]`, а `'Arguments' - sys.argv [3]`. В Python нумерація починається з `0`, а не з `1`.

Команда `sys.path` містить список імен каталогів, звідки імпортуються модулі. Перший рядок в `sys.path` – порожній, показує що поточна директорія також є частиною `sys.path`, яка збігається зі значенням змінної середовища PYTHONPATH. Це означає, що модулі, розташовані в поточному каталозі, можна імпортувати безпосередньо. В іншому випадку доведеться помістити свій модуль в один з каталогів, перерахованих в `sys.path`.

Поточний каталог – це каталог, в якому була запущена програма. Команди `import os; print (os.getcwd ())` – щоб дізнатися поточний каталог програми.

Імпорт модуля – довготривалий процес, тому Python робить деякі кроки для прискорення цього процесу. Один із способів – створити байткод з розширенням `.pyc`, який є проміжною формою, в яку Python переводить програму. Такий файл `.pyc` корисний при імпорті модуля в наступний раз в іншу програму – це відбудеться набагато швидше, оскільки значна частина обробки, необхідної при імпорті модуля, буде вже пророблена. Цей байткод також є від платформи незалежним.

Оператор `from ... import` ... використовується щоб імпортувати змінну `argv` прямо в програму і не писати щоразу `sys`. При зверненні до неї, можна скористатися виразом "`from sys import argv`".

Для імпорту всіх імен, що використовуються в модулі `sys`, можна виконати команду "`from sys import *`". Це працює для будь яких модулів.

У загальному випадку слід уникати використання цього оператора і використовувати замість цього оператор `import`, щоб запобігти конфліктам імен і не ускладнювати читання програми.

```
Приклад:
from math import *
n = input ("Введіть діапазон: - ")
p = [2, 3]
count = 2
a = 5
while (count < n) :
b = 0
for i in range (2, a) :
if ( i < = sqrt(a)) :
if (a % i == 0) :
print ("a neprost", a)
b = 1
else:
pass
if (b ! = 1) :
print ("a prost", a)
p = p + [a]
count = count + 1
```

```
a = a + 2
print p
```

Ім’я модуля – `__name__`. У кожного модуля є ім’я, і команди в модулі можуть дізнатися ім’я їх модуля. Це корисно, коли потрібно знати, чи запущений модуль як самостійна програма або імпортована. Як уже згадувалося вище, коли модуль імпортується вперше, то код що міститься в ньому виконується. Можна скористатися цим для того, щоб змусити модуль поводитися по-різному в залежності від того, чи використовується він сам по собі або імпортується в іншу програму. Цього можна досягти із застосуванням атрибуту модуля під назвою `__name__`.

```
Приклад: (зберегти як using_name.py)
if __name__ == '__main__':
print ('Ця програма запущена сама по собі.')
else:
print ('Мене імпортували в інший модуль.')

Виведення:
$ Python3 using_name.py
Ця програма запущена сама по собі.

$ python3
>>> import using_name
Мене імпортували в інший модуль.
>>>
```

Як це працює. У кожному модулі Python визначено його ім’я – `__name__`. Якщо воно дорівнює `'__main__'`, це означає, що модуль запущений самостійно користувачем, і можна виконати відповідні дії.

7.1. Створення власних модулів

```
Приклад: (зберегти як mymodule.py)
def sayhi ():
print ('Привіт! Це говорить мій модуль.')

__version__ = '0.1'

# Кінець модуля mymodule.py
```

Вище наведено простий модуль. Як видно, в ньому немає нічого особливого в порівнянні зі звичайною програмою на Python. Далі подивимося, як використовувати цей модуль в інших програмах.

Модуль повинен перебувати або в тому ж каталозі, що і програма, в яку імпортується, або в одному з каталогів, зазначених в `sys.path`.

```
Приклад (зберегти як mymodule_demo.py):
import mymodule

mymodule.sayhi ()
print ('Версія', mymodule .__ version__)

Виведення
$ Python mymodule_demo.py
Вітання! Це говорить мій модуль.
версія 0.1
```

Як це працює. Використовуються все ті ж позначення точкою для доступу до елементів модуля. Python повсюдно використовує одне і те ж позначення крапкою, надаючи йому таким чином характерний «Python-овий» вигляд.

```
Синтаксис from..import (зберегти як mymodule_demo2.py):
from mymodule import sayhi, __version__

sayhi ()
print ('Версія', __version__)
```

Висновок `mymodule_demo2.py` такий же, як і `mymodule_demo.py`.

Якщо в модулі, імпортуючий даний модуль, вже було надано ім’я `__version__`, виникне помилка. Це цілком можливо, так як оголошувати версію будь якого модуля за допомогою цього імені – загальноприйнята практика. Тому завжди рекомендується віддавати перевагу оператору `import`, хоча це і зробить програму трохи довшою.

Також можна було б :

```
from mymodule import *
```

Це імпортує всі публічні імена, такі як `sayhi`, але не імпортує `__version__`, тому що воно починається з подвійного підкреслення.

Функція `dir`. Можна використовувати вбудовану функцію `dir`, щоб отримати список ідентифікаторів, які визначає об'єкт. Так в число ідентифікаторів модуля входять функції, класи і змінні, визначені в цьому модулі. Коли передається функції `dir ()` ім'я модуля, вона повертає список імен, визначених в цьому модулі. Якщо ніякого аргументу не передавати, вона поверне список імен, визначених в поточному модулі.

```
Приклад:
$ python3
>>> import sys # отримаємо список атрибутів модуля 'sys'
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__',
'__package__', '__stderr__', '__stdin__', '__stdout__',
'_clear_type_cache', '_compact_freelists', '_current_frames',
'_getframe', 'api_version', 'argv', 'builtin_module_names',
'byteorder', 'call_tracing', 'callstats', 'copyright', 'displayhook',
'dllhandle', 'dont_write_bytecode', 'exc_info', 'excepthook',
'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
'getcheckinterval', 'getdefaultencoding', 'getfilesystemencoding',
'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof',
'gettrace', 'getwindowsversion', 'hexversion', 'intern', 'maxsize',
'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2',
'setcheckinterval', 'setprofile', 'setrecursionlimit', 'settrace',
'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version_info',
'warnoptions', 'winver']

>>> dir () # отримаємо список атрибутів поточного модуля
['__builtins__', '__doc__', '__name__', '__package__', 'sys']

>>> a = 5 # створимо нову змінну 'a'

>>> dir ()
['__builtins__', '__doc__', '__name__', '__package__', 'a',
'sys']

>>> del a # видалимо ім'я 'a'

>>> dir ()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']

>>>
```

Як це працює. Спершу бачимо результат застосування `dir` до імпортованого модулю `sys`. Бачимо величезний список атрибутів, що містяться в ньому.

Потім викликаємо функцію `dir`, не передаючи їй параметрів. За замовчуванням, вона повертає список атрибутів поточного модуля. Список імпортованих модулів також входить туди.

Щоб поспостерігати за дією `dir`, визначаємо нову змінну `a` і присвоюємо їй значення, потім знову викликаємо `dir`. Бачимо, що в отриманому списку з'явилося додаткове значення. Видалимо змінну/атрибут з поточного модуля за допомогою оператора `del`, і зміни знову відобразяться на виведення функції `dir`.

Зауваження з приводу `del`: цей оператор використовується для видалення змінної/імені, і після його виконання, в даному випадку – `del a`, до змінної `a` більше неможливо звернутися – її ніби ніколи й не було.

Функція `dir ()` працює для будь якого об'єкта. Наприклад, виконайте "`dir ('print')`", щоб побачити атрибути функції `print`, або "`dir (str)`", щоб побачити атрибути класу `str`.

7.2. Пакети

Пакети – це просто каталоги з модулями і спеціальним файлом `__init__.py`, який показує Python, що цей каталог особливий, тому що містить модулі Python.

Уявімо, що хочемо створити пакет під назвою «`world`» з субпакетами «`asia`», «`africa`» і т.д., які, в свою чергу, будуть містити модулі «`india`», «`madagascar`» і т.д.

```
Для цього слід було б створити таку структуру каталогів:
| - <деякий каталог із sys.path>/
|      |---- world/
|            |---- __init__.py
|            |---- asia/
|            |      |---- __init__.py
|            |      |---- india/
|            |            |---- __init__.py
|            |            |---- foo.py
|            |---- africa/
|                   |---- __init__.py
|                   |---- madagascar/
|                         |---- __init__.py
|                         |---- bar.py
```

РОЗДІЛ 8. СТРУКТУРИ ДАНИХ

Структури даних – це структури, які можуть зберігати деякі дані разом. Іншими словами, вони використовуються для зберігання пов'язаних даних.

В Python існують чотири вбудованих структури даних: список, кортеж, словник і набір (множини).

8.1. Список (list) в Python

Список – це структура даних яка містить впорядкований набір елементів, тобто можна зберігати послідовність елементів у списку. Це легко уявити, якщо думати про це як про список покупок, який містить речі, які потрібно купити. Список має розміщуватися всередині квадратних дужок. Елементи списку розділяються комою. Список – змінний тип даних. Можна додавати, видаляти і редагувати елементи списку.

Коротке введення в об'єкти і класи. Список – це приклад використання об'єктів і класів. Коли використовується змінна і присвоюється їй значення, скажімо `5`, то можна говорити про це в категоріях об'єктно-орієнтованого програмування – створюється об'єкт (екземпляр) з іменем і класу (типу) `int`. Можна прочитати `help (int)` для того щоб зрозуміти це краще.

Клас також може мати методи – функції визначені для використання тільки по відношенню до цього класу. Можна використовувати ці функції тільки тоді, коли існує об'єкт відповідного класу. Наприклад, Python має спеціальний метод `append` для класу, який дозволяє додавати елемент в кінець списку. Наприклад, `myList.append ('an item')` додасть елемент рядкового типу до списку `myList`. Варто звернути увагу на використання крапкової нотації для доступу до методів об'єктів.

Клас також може містити поля. Поля – це не що інше як змінні визначені для використання тільки зі своїм класом. Та ж історія що і з методами – ці змінні/імена можна використовувати тільки тоді, коли є об'єкт відповідного класу. Для доступу до полів використовується крапкова нотація, наприклад, `myList.field`:

```
Приклад:
#!/usr/bin/python
# Filename: using_list.py
# Це мій список покупок
shoplist = ['apple', 'mango', 'carrot', 'banana']

print ('I have ', len(shoplist), ' items to purchase. ' )

print ('These items are : ', end= ' ' )
for item in shoplist :
print (item, end=' ')

print ('\I also have to buy rice. ' )
shoplist.append ( ' rice ' )
print ('My shopping list is now ', shoplist)

print ('I will sort my list now ' )
shoplist.sort ()
print ('Sorted shopping list is ', shoplist)

print ('The first item I will buy is ', shoplist [0])
olditem = shoplist [0]
del shoplist [0]
print ('I bought the ', olditem)
print ('My shopping list is now ', shoplist)

Виведення:
$ python using_list.py
I have 4 items to purchase.
These items are: apple mango carrot banana
I also have to buy rice.
My shopping list is now ['apple', 'mango', 'carrot', 'banana',
'rice']
I will sort my list now
Sorted shopping list is ['apple', 'banana', 'carrot', 'mango',
'rice']
The first item I will buy is apple
I bought the apple
My shopping list is now ['banana', 'carrot', 'mango', 'rice']
```

Як це працює. Змінна `shoplist` це список покупок для того хто збирається на ринок. У `shoplist` зберігається назви того, що треба купити. Можна додати об’єкт будь якого типу до списку, включаючи числа і навіть, інші списки.

Також використано цикл `for…in` для ітерації по елементам списку. На даний момент треба зрозуміти, що списки це також і послідовність.

Варто звернути увагу на використання іменованого аргументу `end` в функції `print`. Його тут було використано для того, щоб вказати на те що замість традиційного символу нового рядка потрібен символ пробілу в кінці виведення.

Далі додано елемент в кінець списку користуючись методом `append` об'єкту типу «список». Тоді перевіряємо чи елемент насправді було додано до списку вивівши його вміст на екран функцією print.

Далі, список був відсортований методом `sort`. Важливо розуміти що список було відсортовано на місці, тобто даний метод змінює сам список переданий йому. Він не повертає новий відсортований список. Це відрізняється від того як працюють рядки. Тобто це означає, що списки можна змінювати, а рядки ні.

Коли вже куплено щось зі списку, то потрібно це щось видалити з нього. Для цього необхідно використати інструкції `del`. Оскільки потрібно видалити перший елемент списку, то пишемо `del shoplist [0]`.

8.2. Кортеж (tuple) в Python

Головна відмінність між кортежем і списком полягає в тому, що кортеж є незмінним. Не можна змінити кількість елементів кортежу, чи наприклад, певний елемент кортежу. Крім того, для кортежу потрібно використовувати дужки – `()` замість `[]`. Елементи кортежу відокремлюються комами.

Кортеж зазвичай використовується у тих випадках, коли інструкція чи визначена користувачем функція, може безпечно припустити, що елементи кортежу не зміняться.

```
Приклад:
#!/usr/bin/python
# Filename: using_tuple.py
zoo = ('python', 'elephant', 'penguin') # дужки необов'язкові
print ('Number of animals in the zoo is ', len(zoo))

new_zoo = ('monkey', 'camel', zoo)
print ('Number of cages in the new zoo is ', len (new_zoo))
print ('All animals in new zoo are ', new_zoo)
print ('Animals brought from old zoo are ', new_zoo [2])
print ('Last animal brought from old zoo is ', new_zoo [2][2])
```

```
    print ('Number of animals in the new zoo is ', len ( new_zoo) -
1+len ( new_zoo [2]))

    Виведення:
    $ python using_tuple.py
    Number of animals in the zoo is 3
    Number of cages in the new zoo is 3
    All animals in new zoo are ('monkey', 'camel', ('python',
    'elephant', 'penguin'))
    Animals brought from old zoo are ('python', 'elephant',
    'penguin')
    Last animal brought from old zoo is penguin
    Number of animals in the new zoo is 5
```

Як це працює. Змінна `zoo` посилається на кортеж. Функція `len` може бути використана і з кортежем. Переміщаємо цих тварин в новий зоопарк (змінна `new_zoo`) оскільки старий зоопарк закрився. Таким чином кортеж `new_zoo` містить тих тварин які були там з моменту створення і тварин зі старого зоопарку. Назад в реальність, варто звернути увагу на те що кортеж всередині іншого кортежу залишається кортежем.

Доступ до елементів кортежу можна отримати вказуючи позицію (індекс) елементу всередині квадратних дужок. Це називається індексним оператором. Доступ до третього елементу `new_zoo` отримано за допомогою конструкції: `new_zoo[2]`. А для того щоб вказати третій елемент всередині третього елементу кортежу `new_zoo new_zoo[2][2]`. Така конструкція стане доволі простою як тільки урозуміється ідіома (Ідіома – стійкий неподільний зворот мови, що передає єдине поняття, зміст якого не визначається змістом його складових елементів).

Дужки. Не зважаючи на те що дужки є необов'язковими, їх завжди використовують для того, щоб явно вказати на те що це є кортеж. Наприклад, `print (1,2,3)` і `print ( (1,2,3) )` означають дві різні речі — перша інструкція виводить на екран три різних числа, друга виводить кортеж (який містить три числа).

Кортеж без елементів або з 1 елементом. Порожній кортеж можна створити просто вказавши пару порожніх дужок: `empty = ()`. Проте кортеж з одним елементом не такий простий. Для того щоб створити такий кортеж треба вказати кому, яка слідує за першим (і єдиним) його елементом для того щоб Python міг

відрізнити кортеж від об’єкту оточеного дужками який використовується в якомусь виразі. Кортеж з одним елементом: `singleton = (2 ,)`.

8.3. Словники в Python

Словник це ніби адресна книга де можна знайти адресу або контактні дані певної особи знаючи тільки її/його ім’я. Іншими словами асоціюємо ключі (імена) із значеннями (контактними даними). Варто зауважити, що ключ має бути унікальним інакше не можливо знайти потрібну інформацію, якщо є дві особи з абсолютно однаковими іменами.

В якості ключів для словника можуть виступати тільки незмінні об’єкти (такі як рядки). Але це правило не стосується значень словника.

Пари ключ-значення вказуються в словнику використовуючи наступну нотацію: `словник = {ключ1 : значення1, ключ2 : значення2}`. Пари ключ-значення розділені двокрапкою, а самі пари розділяються комами.

Пари ключ-значення жодним чином не впорядковані. Якщо потрібно, щоб вони йшли в певному порядку, то доведеться самостійно їх відсортувати перед використанням.

Приклад:

```
#!/usr/bin/python
# Filename: using_dict.py
# ' ab ' скорочення від адресної книги
ab = {'Swaroop ' : ' swaroops.com ',
'Larry ' : ' larry-wall.org ',
'Matsumoto ' : ' matz-ruby-lang.org ',
'Spammer ' : ' spammer.com '
}

print("Swaroop's address is", ab ['Swaroop'])

# Видалення пари ключ-значення
del ab ['Spammer']

print ('\nThere are {0} contacts in the address-book\n '
.format(len(ab)))

for name, address in ab.items():
print ('Contact {0} at {1} '.format (name, address))
```

```
# Додавання нової пари
ab ['Guido'] = ' guido-python.org '

#Додаємо до словника нову пару ключ-значення
if 'Guido' in ab : # Або ab.has_key ('Guido')
print ( " \nGuido's address is ", ab ['Guido'])

Виведення:
$ python using_dict.py
Swaroop's address is swaroops.com
There are 3 contacts in the address-book
Contact Swaroop at swaroops.com
Contact Matsumoto at matz-ruby-lang.org
Contact Larry at larry-wall.org
Guido's address is guido-python.org
```

Як це працює. Створено словник `ab` використовуючи відповідний синтаксис. Потім отримано доступ до пар ключ-значення використавши індексний оператор.

Можна видаляти пари ключ-значення використовуючи інструкцію `del`. Для цього лише потрібно вказати потрібний ключ потрібного словника.

Далі, скориставшись методом `items` словника (даний метод повертає список кортежів в якому кожен кортеж складається з пари елементів – ключа і значення), отримали доступ до кожної пари ключ-значення. Отримавши цю пару і привласнивши її значення відповідним змінним `name` і `address` використовуючи цикл `for…in`. І потім вивели на екран ці значення всередині циклу `for`.

Для того, щоб додати нову пару ключ-значення до словника достатньо просто вказати новий унікальний ключ, і тут же привласнити йому значення.

Для перевірки того, чи існує потрібна пара ключ-значення можна скористатися оператором `in` або методом `has_key` класу `dict`. Можна переглянути документацію, для того щоб дізнатися повний список методів, виконавши команду `help (dict)`.

8.4. Послідовності в Python

Списки, кортежі і рядки є прикладами послідовностей, але що таке послідовності і що в них особливого?

Серед головних можливостей наявність тестів на входження (вирази `in` та `not in`) і операції індексування. Операція індексування дозволяє отримати певний елемент в послідовності напряму.

Три типи послідовностей згаданих вище – списки, кортежі і рядки, також підтримують операцію зрізу, яка дозволяє отримати зріз послідовності (її частину).

```
Приклад:
#!/usr/bin/python
# Filename: seq.py

shoplist = ['apple', 'mango', 'carrot', 'banana']
name = 'swaroop'

# Операція індексування
print ('Еелемент 0 це ', shoplist [0])
print ('Еелемент 1 це ', shoplist [1])
print ('Еелемент 2 це ', shoplist [2])
print ('Еелемент 3 це ', shoplist [3])
print ('Еелемент -1 це ', shoplist [-1])
print ('Еелемент -2 це ', shoplist [-2])
print ('Символ 0 це ', name [0])

# Зріз списку
print ('Еелементи з 1 по 3 це ', shoplist [1:3])
print ('Еелементи з 2 до кінця це ', shoplist [2:])
print ('Еелементи з 1 по -1 це ', shoplist [1:-1])
print ('Еелементи з початку до кінця це ', shoplist [:])

# Зріз рядка
print ('Символи з 1 по 3 це ', name [1:3])
print ('Символи з 2 до кінця це ', name [2:])
print ('Символи з 1 по -1 це ', name [1:-1])
print ('Символи з початку до кінця це ', name[:])

Виведення:
$ python seq.py
Еелемент 0 це apple
Еелемент 1 це mango
Еелемент 2 це carrot
Еелемент 3 це banana
Еелемент -1 це banana
Еелемент -2 це carrot
Символ 0 це s
Еелементи з 1 по 3 це ['mango', 'carrot']
Еелементи з 2 до кінця це ['carrot', 'banana']
Еелементи з 1 по -1 це ['mango', 'carrot']
Еелементи з початку до кінця це ['apple', 'mango', 'carrot',
'banana']
Символи з 1 по 3 це wa
```

```
Символи з 2 до кінця це aroop
Символи з 1 по -1 це waroo
Символи з початку до кінця це swaroop
```

Як це працює. Індекси використовуються для того щоб дістатися до окремих елементів послідовності. Щоразу коли вказується число для послідовності всередині квадратних дужок, як в програмі вище, Python поверне елемент який відповідає вказаній позиції всередині послідовності.

Індексом також може бути негативне число. У цьому випадку позиція вираховується з кінця послідовності. Отже, `shoplist [-1]` посилається на останній елемент послідовності, а `shoplist [-2]` вибирає другий елемент з кінця (передостанній).

Операція зрізу використовується вказуючи ім'я послідовності за якою слідують необов'язкові два числа розділені двокрапкою всередині квадратних дужок. Це схоже на операцію індексування.

Перше число (перед двокрапкою) в операції зрізу позначає позицію початку зрізу, а друге число вказує на кінець зрізу. Якщо початок зрізу не вказувати явно, то Python почне з 1-го елементу послідовності. Якщо пропущено друге число після двокрапки, то Python закінчить операцію зрізу на останньому елементі. Нижня межа зрізу буде починатися саме з 1-го вказаного числа (число перед двокрапкою), але верхня межа, закінчиться якраз перед другим вказаним індексом послідовності. Іншими словами, стартова позиція включається у зріз, але остання позиція виключається зі зрізу.

Таким чином, `shoplist [1:3]` поверне зріз послідовності починаючи з позиції `1`, включаючи позицію `2`, але зупиниться на позиції `3`, і тому, повернеться зріз з `2` елементів. Схожим чином, `shoplist [:]` поверне копію всієї послідовності.

Існує можливість робити зрізи використовуючи від'ємні числа. Числа зі знаком мінус використовуються для позицій з кінця послідовності. Наприклад, `shoplist [:-1]` поверне зріз послідовності в якому немає останнього елемента послідовнсті, але присутні всі інші елементи.

Також можна вказати третій параметр для послідовності, який буде вказувати на крок для зрізування (типово, крок дорівнює 1):

```
>>> shoplist = ['apple', 'mango', 'carrot',' banana']
>>> shoplist [::1]
['apple', 'mango', 'carrot', 'banana']
>>> shoplist [::2]
['apple', 'carrot']
>>> shoplist[::3]
['apple', 'banana']
>>> shoplist [::-1]
['banana', 'carrot', 'mango', 'apple']
```

Коли крок дорівнює 2, то отримуємо елементи з позицією `0, 2,…` Коли ж крок дорівнює трьом, то елементи з позицією `0, 3, etc`.

8.5. Набір (множина) в Python

Набори (множини) – це невпорядковані колекції простих об’єктів. Вони використовуються, тоді коли існування об’єктів в колекції є більш важливим ніж порядок їхнього розташування, чи те скільки разів вони зустрічаються.

Використовуючи набори маємо можливість перевіряти приналежність (членство) тих чи інших об’єктів до множини, визначати те чи знаходиться один набір всередині іншого, знаходити перетин між двома наборами, і т.д.

```
>>> bri = set (['brazil', 'russia', 'india'])
>>> 'india' in bri
True
>>> 'usa' in bri
False
>>> bric = bri.copy ()
>>> bric.add ('china')
>>> bric.issuperset (bri)
True
>>> bri.remove ('russia')
>>> bri & bric #Або bri.intersection (bric)
{'brazil', 'india'}
```

Як це працює. Вищенаведений приклад зрозумілий без пояснень, тому що в ньому використовується базова теорія множин з курсу математики, вивчена в школі.

8.6. Посилання в Python

Коли створюється об’єкт і присвоюється його якійсь змінній, вона лише посилається на об’єкт, але не представляє власне сам об’єкт. Саме так, ім’я змінної вказує на ту частину пам’яті комп’ютера, де зберігається об’єкт. Це називається прив’язуванням імені до об’єкта.

```
Приклад:
#!/usr/bin/python
# Filename: reference.py

print ('Simple assignment')
shoplist = ['apple', 'mango', 'carrot', 'banana']
mylist = shoplist # mylist — це просто інше ім’я яке вказує на
об’єкт shoplist!

del shoplist[0] # Я придбав перший елемент у списку, тому видаляю
його

print ('shoplist is ', shoplist)
print ('mylist is ', mylist)
# shoplist і mylist містять один і той самий список
# без 'apple', таким чином підтверджуючи те, що вони вказують на
один об’єкт

print ('Copy by making a full slice')
mylist = shoplist[:] # копіювання шляхом повного зрізу

del mylist[0] # видалити 1-й елемент

print ('shoplist is ', shoplist)
print ('mylist is ', mylist)
# варто звернути увагу на те що тепер, два списки різні

Виведення:
$ python reference.py
Simple Assignment
shoplist is ['mango', 'carrot', 'banana']
mylist is ['mango', 'carrot', 'banana']
Copy by making a full slice
shoplist is ['mango', 'carrot', 'banana']
mylist is ['carrot', 'banana']
```

Як це працює. Якщо потрібно зробити копію списку чи чогось подібного, чи складних об’єктів, тоді можна використати оператор зрізу. Якщо просто присвоєно ім’я змінної іншій змінній, то вони будуть просто посилатися на один і той же об’єкт, що може спричинити проблеми.

РОЗДІЛ 9. СТВОРЕННЯ ПРОГРАМИ В Python

У попередніх розділах досліджено різноманітні частини мови Python, однак варто глянути як усі ці частини поєднуються разом, розробивши програму, яка робить щось корисне. Ідея полягає в тому, щоб самотужки навчитися писати скрипти на Python.

Постановка задачі. Розробка резервної копії усіх існуючих важливих файлів.

Дану задачу можна розв'язати різними способами. Вподальшому розглянемо декілька з них.

Незважаючи на те, що це проста програма і недостатньо інформації для того, щоб почати розв'язок. Потрібно аналіз задачі. Наприклад:

- Як вказати для яких саме файлів потрібно створювати резервні копії?
- Як вони будуть зберігатися?
- Де вони будуть зберігатися?

Після правильно проведеного аналізу, починаємо проектувати (планувати) програму. Складаємо алгоритм того, як програма має працювати.

1. Перелік файлів і каталогів для копіювання вказуються як список.
2. Резервна копія повинна зберігатися в головному каталозі для резервних копій.
3. Резервні копії створюються у вигляді zip файлів.
4. Назва zip архіву – поточна дата і час.
5. Скористаємося стандартною командою zip, яка типово доступна в стандартних інсталяціях дистрибутивів Linux/Unix, Windows.

9.1. Розв'язок № 1 задачі

Оскільки проект майбутньої програми більш-менш узгоджено можемо почати писати код, який є втіленням рішення.

```
#!/usr/bin/python
# Filename: backup_ver1.py

import os
```

```
import time

# 1. Файли і каталоги для резервного копіювання вказуються як
список
source = ['"C:\\My Documents"', 'C:\\Code']
# нам довелося використати подвійні лапки всередині рядка для
імен з пробілами

# 2. Резервна копія буде зберігатися в головному каталозі для
резервних копій
target_dir = 'E:\\Backup' # Не забудь змінити шлях відповідно до
твоїх потреб

# 3. Файли зберігаються як архів формату zip
# 4. Назва архіву — це поточна дата і час
target = target_dir + os.sep + time.strftime('%Y%m%d%H%M%S') +'.z
ip'

# 5. Використовуємо команду zip для архівації файлів
zip_command = "zip -qr {0} {1}".format (target, ' '.join(source))

# Запустити процес резервного копіювання
if os.system(zip_command) == 0:
print ('Дані збережено до ', target)
else:
print ('Процес закінчився НЕВДАЧЕЮ')

Виведення:
$ python backup_ver1.py
Дані збережено до E:\Backup\20080702185040.zip
```

Перейшли до стадії тестування. Перевіряємо чи програма працює правильно. Якщо вона не поводиться як очікували, тоді потрібно відлагодити програму. Іншими словами, усунути помилки (баги).

Якщо вищенаведений код не працює, потрібно написати `print (zip_command)` якраз перед `os.system` і запусти програму. Далі потрібно скопіювати вставити, те що було виведено на екран в результаті виконання команди `print (zip_command)`, в командну оболонку Python та подивитись чи код виконується сам по собі. Якщо ні, то перевірити керівництво до команди `zip`. Якщо ж виконання команди було успішним, то тоді перевірити код програми.

Як це працює. Було використано модулі `os` і `time` спершу імпортувавши їх. Потім створено список `source`, де вказано для яких папок і каталогів мають бути створені резервні копії. Змінна `target_dir` містить шлях до каталога де файли

будуть збережені. Ім’я архіву, який планується створити, складається з поточною датою і часом, при використані функції `time.strftime()`. Кінцевий файл матиме розширння `zip` і буде зберігатися в каталозі `target_dir`.

Варто звернути увагу на змінну `os.sep`. Вона містить роздільник директорій, який зважаючи на ОС в якій запущено програму, буде різним. Для Linux і Unix це буде символ `'/'`, для Windows символ '\\', Mac OS же використовує `':'`. Завдяки використанню цієї змінної програма буде працювати на всіх ОС без змін в програмному коді.

Функція `time.strftime ()` приймає рядок певного вигляду (специфікації), такий як використано в програмі вище. Символ `%Y` буде замінений на поточний рік без століття. Символ `%m` — на номер поточного місяця у вигляд `01`, `02`, `12` і т.д. Повний список всіх специфікторів можна знайти в `help ()`.

Створили ім’я для цільового `zip` файлу, використовуючи оператор додавання, який конкатенує рядки разом і повертає новий рядок. Потім створено змінну рядкового типу `zip_command`, яка містить команду, яку буде виконано. Для того, щоб переконатися, що ця команда працює варто виконати її в командній оболонці (Linux термінал чи командний рядок DOS).

Команда `zip` має певні параметри і опції. `-q` вказує на те що команда `zip` має працювати тихо (жодних повідомлень на екран не буде виводитися). `-r` означає, що програма буде працювати рекурсивно по відношенню до каталогів, тобто вона пройде по всім каталогам і їх підкаталогам а також, файлам які там є. Дві вищевказані опції можна поєднати: `-qr`. Метод `join` використано для конвертації списку `source` в рядок.

Потім запускаємо команду, скориставшись функцією `os.system`. Вона запускає команду так ніби вона була запущена прямо з ОС, в командному рядку – функція поверне `0`, якшо завершилася успішно, інакше, в програму повернеться код помилки.

В залежності від виведення на екран відповідне повідомлення про те була операція успішною чи ні.

9.2. Розв’язок №2 задачі

1-а версія скрипта працює. Тим не менше, можна провести деякі вдосконалення, щоб програма працювала краще. Це називається фазою супроводу програми.

Одне із вдосконалень, яке буде корисним це кращий механізм іменування файлів – використовуючи час як ім’я файлу всередині каталога і поточну дату для іменування каталогу всередині головного резервного каталогу. Перша перевага полягає в тому що резервні копії будуть зберігатися в ієрархічному порядку і таким чином, орієнтуватися у всіх цих файлах буде набагато легше. Наступна перевага – імена файлів стануть набагато коротшими. Третя перевага – легко перевірити чи є бекап для кожного дня.

```
#!/usr/bin/python
# Filename: backup_ver2.py

import os
import time

# 1. Каталоги і файли для яких треба створити резервні копії перераховуються в списку
source = ['"C:\\My Documents"', 'C:\\Code']
# Зауваження, ми змушені були використовувати подвійні лапки всередині рядка для імені з пробілами

# 2. Копія має зберігатися в головному каталозі
target_dir = 'E:\\Backup' # Не забудь змінити шлях на власний

# 3. Файли зберігаються в zip архіві
# 4. Поточний день — назва для підкаталогу в головному каталозі

today = target_dir + os.sep + time.strftime('%Y%m%d')
# Поточний час — назва для zip архіву
now = time.strftime ('%H%M%S')

# Створити підкаталог, якщо його ще не існує
if not os.path.exists(today):
    os.mkdir (today) # створити каталог
    print ('Каталог успішно створено', today)

# Ім’я zip файлу
target = today + os.sep + now + '.zip'

# 5. Використовуємо команду zip для створення архіву
zip_command = "zip -qr {0} {1}".format (target, ' '.join(source))
```

```
# Запустити процес
if os.system(zip_command) == 0:
    print ('Дані збережено до', target)
else:
    print ('Процес закінчився невдачею')

Виведення:

$ python backup_ver2.py
Каталог успішно створено E:\Backup\20080702
Дані збережено до E:\Backup\20080702\202311.zip

$ python backup_ver2.py
Дані збережено до E:\Backup\20080702\202325.zip
```

Як це працює. Більша частина коду залишилася незмінною. Зміни полягають в перевірці існування каталогу з поточним днем в якості назви всередині головного каталогу. З цією метою використано функцію `os.path.exists`. Якщо каталогу з таким іменем не існує, то його створюємо функцією `os.mkdir`.

9.3. Розв'язок №3 задачі

Друга версія програми працює добре, коли створено багато резервних копій. Але, коли їх стає дуже багато, стає важко відрізнити які з них для чого були створені. Наприклад, зроблено якісь значні зміни в програмі або презентації, тому варто вказати, що це були за зміни в імені `zip` архіву. Цього легко досягти, додавши коментарі користувача до назви архіву.

Зауваження
Наступний код не працює, тому не варто лякатись, це буде уроком.

```
#!/usr/bin/python
# Filename: backup_ver3.py

import os
import time

# 1. Каталоги і файли для яких треба створити резервні копії
перераховуються в списку
source = ['"C:\\My Documents"', 'C:\\Code']
# Зауваження, ми змушені були використовувати подвійні лапки
всередині рядка для імен з пробілами

# 2. Копія має зберігатися в головному каталозі
target_dir = 'E:\\Backup' # Не забудь змінити шлях на власний
```

```
# 3. Файли зберігаються в zip архіві
# 4. Поточний день — назва для підкаталогу в головному каталозі

today = target_dir + os.sep + time.strftime('%Y%m%d')
# Поточний час — назва для zip архіву
now = time.strftime('%H%M%S')

# Прийняти коментар від користувача для створення імені архіву
comment = input('Введи коментар --> ')
if len(comment) == 0: # перевіряємо чи коментар було введено
target = today + os.sep + now + '.zip'
else:
target = today + os.sep + now + '_' +
comment.replace(' ', '_') + '.zip'

# Створити підкаталог, якщо його ще не існує
if not os.path.exists(today):
os.mkdir(today) # створити каталог
print ('Каталог успішно створено', today)

# 5. Використовуємо команду zip для створення архіву
zip_command = "zip -qr {0} {1}".format (target, ' '.join(source))

# Запустити процес
if os.system(zip_command) == 0:
print ('Дані збережено до', target)
else:
print ('Процес закінчився НЕВДАЧЕЮ')

Виведення:
$ python backup_ver3.py
    File "backup_ver3.py", line 25
        target = today + os.sep + now + '_' +
                                                ^
SyntaxError: invalid syntax
```

Як це (не працює) працює. Ця програма не працює. Python говорить про синтаксичну помилку, що значить, що структура скрипта не задовільняє Python. Повідомлення про помилку, крім всього іншого, також вказує на місце де саме вона сталася. Отже, варто відлагодити (`debugging`) програму з вказаного рядка.

Уважно придивившись до коду бачимо, що логічний рядок розташований на двох фізичних рядках, але не вказано, що ці два фізичних рядки одна інструкція. В тому місці Python знайшов оператор + без операнду і тому не знає, що робити далі. Можна вказати, що логічний рядок продовжується на наступному фізичному рядку за допомогою оберненої бек флеш (`\`) розташованої наприкінці фізичного рядка.

Отже, внесено поправки до коду. Корекція коду, у випадку знаходження помилок, називається виправленням помилок.

9.4. Розв'язок №4 задачі

```
#!/usr/bin/python
# Filename: backup_ver4

import os
import time

# 1. Каталоги і файли для яких треба створити резервні копії перераховуються в списку
source = ['"C:\\My Documents"', 'C:\\Code']
# Зауваження, ми змушені були використовувати подвійні лапки всередині рядка для імен з пробілами

# 2. Копія має зберігатися в головному каталозі
target_dir = 'E:\\Backup' # Не забуть змінити шлях на власний

# 3. Файли зберігаються в zip архіві
# 4. Поточний день — назва для підкаталогу в головному каталозі
today = target_dir + os.sep + time.strftime('%Y%m%d')
# Поточний час — назва для zip архіву
now = time.strftime('%H%M%S')

# Прийняти коментар від користувача для створення імені архіву
comment = input('Введи коментар --> ')
if len(comment) == 0: # перевіряємо чи коментар було введено
target = today + os.sep + now  + '.zip'
else:
target = today + os.sep + now + '_' + \
comment.replace(' ', '_') + '.zip'

# Створити підкаталог, якщо його ще не існує
if not os.path.exists(today):
os.mkdir(today) # створити каталог
print ('Каталог успішно створено', today)

# 5. Використовуємо команду zip для створення архіву
zip_command = "zip -qr {0} {1}".format (target, ' '.join(source))

# Запустити процес
if os.system(zip_command) == 0:
print ('Дані збережено до', target)
else:
print ('Процес закінчився НЕВДАЧЕЮ')

Виведення:
$ python backup_ver4.py
Введи коментар --> added new examples
```

```
Дані збережено до
E:\Backup\20080702\202836_added_new_examples.zip

$ python backup_ver4.py
Введи коментар -->
Дані збережено до E:\Backup\20080702\202839.zip
```

Як це працює. Тепер програма працює. Коментарі від користувача отримуємо за дописуючи функції `input`. Потім перевіряємо чи веде щось шляхом перевірки довжини рядка (функція `len`). Якщо ж просто натиснути клавішу Enter, не вводячи жодного тексту (можливо це була звичайна копія, без жодних особливих змін), тоді продовжуємо як і раніше.

Однак, якщо коментар було введено, тоді він стає частиною імені `zip` архіву. Замінюємо пробіли символами нижнього підкреслювання – це тому що керувати файлами без пробілів в іменах набагато легше.

Четверта версія працює задовільно для більшості користувачів, але завжди є простір для вдосконалень. Наприклад, можна вказувати рівень інформативності (через опції `-v` в `CLI`) для того, щоб зробити програму більш «балакучою» (будуть виводитися повідомлення які вказують на те, що програма робить).

Інше можливе поліпшення – можливість вказати в командному рядку додаткові каталоги і файли для резервного копіювання. Можна отримувати їхні назви зі списку `sys.argv`, потім додавати до змінної `source` використовуючи метод `extend` класу `list`.

Найбільш важливим поліпшенням програми було б використання модуля `zipfile` або `tarfile` замість методу з `os.system`. Ці модулі є частиною стандартної бібліотеки і можна позбутися від зовнішньої залежності у вигляді програми `zip`.

Проте використано метод з `os.system` чисто з педагогічних міркувань – приклад вийшов достатньо простим для розуміння і в той же час реалістичним.

9.5. Процес розробки програмного забезпечення

До цього моменту пройшли через різні фази в процесі написання програм. Ці фази можна підсумувати у вигляді наступного списку:

1. Що (Аналіз);

2. Як (Планування, проектування);

3. Зроби це (Втілення в коді, написання програми);

4. Тест (Тестування і виправлення помилок);

5. Використання (Розгортання);

6. Супровід (Вдосконалення).

Рекомендований спосіб написання програм – це процедура, якій слідували при створенні скрипта для резервного копіювання: спершу провели аналіз і спроектували майбутню програму, потім почали писати программу, спершу створивши просту її версію, далі провели тестування і виправлення помилок. Переконалися, що програма працює як потрібно. В кінці додали нові можливості. Якщо є необхідність допрацьовувати програму, то потрібно буде повторювати цей цикл – написати, протестувати, спробувати стільки разів скільки буде потрібно.

Додаток 1

Оператори та їх застосування

Оператор	Назва	Пояснення	Приклади
+	Додавання	Додає об’єкти	3 + 5 рівне 8; 'a' + 'b' рівне 'ab'
-	Віднімання	Дає різницю двох чисел; якщо перший операнд відсутній, він вважається рівним нулю	-5.2 маємо від’ємне число, а 50 - 24 маємо 26.
*	Множення	Дає добуток двох чисел або повертає рядок, заданим числом раз.	2 * 3 маємо 6. 'la' * 3 маємо 'lalala'.
**	Піднесення до степення	Повертає число x, зведена в ступінь y	3 ** 4 маємо 81 (тобто 3 * 3 * 3 * 3)
/	Ділення	Повертає частку від ділення x на y	4 / 3 маємо 1.3333333333333333.
//	Цілочисельне ділення	Повертає неповну частку від ділення	4 // 3 маємо 1.
%	Ділення по модулю	Повертає залишок від ділення	8 % 3 маємо 2. -25.5 % 2.25 маємо 1.5.
<<	Зрушення вліво	Зміщує біти числа вліво на задану кількість позицій. (Будь-яке число в пам’яті комп’ютера представлено у вигляді бітів -або двійкових чисел, тобто 0 і 1)	2 << 2 маємо 8. У двійковому вигляді 2 являє собою 10. Зрушення вліво на 2 біта дає 1000, що в десятковому вигляді означає 8.
<<	Зрушення вправо	Зміщує біти числа вправо на задане число позицій.	11 >> 1 маємо 5. У двійковому вигляді 11 представляється як 1011 що будучи зміщеним на 1 біт вправо, дає 101, а це, в свою чергу, не що інше як десяткове 5
&	Побітове “і”	Побітова операція “і” над числами	5 & 3 маємо 1.
|	Побітове “або”	Побітова операція “або” над числами	5 | 3 маємо 7
^	Побітове виключення	Побітова операція “виключно або”	5 ^ 3 маємо 6
~	Побітове “не”	Побітова операція “не” для числа x відповідає - (x + 1)	~5 маємо -6.
<	Менше	Визначає, чи вірно, що x менше y. Всі оператори порівняння повертають True або False. Зверніть увагу на великі літери в цих словах.	5 <3 маємо False, а 3 <5 дасть True. Можна складати довільного ланцюжка порівнянь: 3 <5 <7 дає True.
>	Більше	Визначає, чи вірно, що x більше y	5> 3 дає True. Якщо обидва операнда - числа, то перед порівнянням вони обидва переходять до однакового типу. В іншому випадку завжди повертається False.
<=	Менше або рівне	Визначає, чи вірно, що x менше або дорівнює y	x = 3; y = 6; x <= y маємо True.
>=	Більше або рівне	Визначає, чи вірно, що x більше або дорівнює y	x = 4; y = 3; x >= 3 маємо True.
==	Рівне	Перевіряє, чи однакові об'єкти	x = 2; y = 2; x == y маємо True. x = 'str'; y = 'stR'; x == y маємо False. x = 'str'; y = 'str'; x == y маємо True.
!=	Не рівне	Перевіряє, чи вірно, що об'єкти не рівні	x = 2; y = 3; x != y маємо True.

not	Логічне не	Якщо x дорівнює True, оператор поверне False. Якщо ж x дорівнює False, отримаємо True.	x = True; not x маємо False.
and	Логічне і	x and y дає False, якщо x дорівнює False, в іншому випадку повертає значення y	x = False; y = True; x and y повертає False, оскільки x дорівнює False. В цьому випадку Python не стане перевіряти значення y, так як вже знає, що ліва частина виразу 'and' дорівнює False, що має на увазі, що і весь вираз в цілому дорівнюватиме False, незалежно від значень всіх інших операндів. Це називається скороченою оцінкою булевих (логічних) виразів.
or	Логічне або	Якщо x дорівнює True, в результаті отримаємо True, в іншому випадку отримаємо значення y	x = True; y = False; x or y дає True. Тут також може проводитися укорочена оцінка виразів.

«True» - англ. «Правда»; «False» - англ. «Ложь».

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Swaroop C. H. A Byte of Python (Russian) Version 2.01 / Swaroop C. H.; Translated by Vladimir Smolyar. – Moscow, 2013. – 151 с.

2. Доусон М. Программируем на Python / Доусон М. – 3-е изд. – СПб. : Питер, 2014. – 416 с.

3. Чан Уэсли Дж. Python: создание приложений / Чан Уэсли Дж. – 3-е изд. – М. : Вильямс, 2015. – 816 с.

4. Рубанцев В. Решение задач на языке Python 3.x (Сокращённый вариант) / Рубанцев В. – М. : RVGames, 2014. – 110 с.

5. Пилгрим Марк. Погружение в Python 3 (Dive into Python 3 на русском) / Пилгрим Марк. – М. : Самиздат. – 2009. – 148 с.

Навчальне видання

Замуруєва Оксана Валеріївна
Кримусь Андрій Сергійович
Ольхова Наталя Володимирівна

Об’єктно-орієнтоване програмування в Python

Курс лекцій

Друкується в авторській редакції