Посібник з програмування на Python

- Запис та читання архівних zip-файлів
- Розділ 10. Робота з датами та часом
  - Модуль datetime
  - Операции с датами

Розділ 1. Введення в Python

Мова програмування Python

Python представляє популярну високорівневу мову програмування, яка призначена для створення додатків різних типів. Це і веб-програми, і ігри, і настільні програми, і робота з базами даних. Досить велике поширення пітон отримав у галузі машинного навчання та досліджень штучного інтелекту.

Вперше мова Python була анонсована 1991 року голландським розробником Гвідо Ван Россумом. З того часу ця мова пройшла великий шлях розвитку. У 2000 році було видано версію 2.0, а в 2008 році - версію 3.0. Незважаючи на такі великі проміжки між версіями постійно виходять підверсії. Так, поточною актуальною версією на момент написання цього матеріалу є 3.13 , яка вийшла у жовтні 2024 року.

Основні особливості мови програмування Python:

- Скриптова мова. Код програм визначається як скриптів.
- Підтримка різних парадигм програмування, у тому числі об'єктно-орієнтованої та функціональної парадигм.
- Інтерпретація програм. Для роботи зі скриптами необхідний інтерпретатор, який запускає та виконує скрипт.

  Виконання програми на Python виглядає так. Спочатку ми пишемо в текстовому редакторі скрипт з набором виразів цією мовою програмування. Передаємо цей скрипт виконання інтерпретатору. Інтерпретатор транслює код у проміжний байткод, а потім віртуальна машина переводить отриманий байткод на набір інструкцій, що виконуються операційною системою.

  Тут варто зазначити, що хоча формально трансляція інтерпретатором вихідного коду в байткод і переведення байткоду віртуальною машиною в набір машинних команд представляють два різні процеси, але фактично вони об'єднані в інтерпретаторі.

   Виконання програми на Python
- Портативність та платформонезалежність. Не має значення, яка у нас операційна система – Windows, Mac OS, Linux, нам достатньо написати скрипт, який запускатиметься на всіх цих ОС за наявності інтерпретатора
- Автоматичне керування пам'яті
- Динамічна типізація

Python - дуже проста програма, вона має короткий і в той же час досить простий і зрозумілий синтаксис. Відповідно його легко вивчати, і власне це одна з причин, через яку він є однією з найпопулярніших мов програмування саме для навчання. Зокрема, у 2014 році він був визнаний найпопулярнішою мовою програмування для навчання у США.

Python також популярний не тільки у сфері навчання, а й у написанні конкретних програм у тому числі комерційного характеру. Немалою мірою тому для цієї мови написано багато бібліотек, які ми можемо використовувати.

Крім того, у цієї мови програмування дуже велике співтовариство програмістів, в інтернеті можна знайти з цієї мови безліч корисних матеріалів, прикладів, отримати кваліфіковану допомогу фахівців.

Пакети та бібліотеки

Інтерпретатор Python супроводжується достатнім функціоналом, який дозволяє створювати програми цією мовою. Проте цього функціоналу може виявитися замало низки завдань. Але через велику спільноту розробників цією мовою по всьому світу також є велика екосистема різних пакетів і бібліотек, які можна використовувати для різних цілей. У розділі мови Python на сайті METANIT.COM будуть розглянуті деякі з цих бібліотек. Перелічу основні їх.

Для створення графічних програм:

- Tkinter
- PyQt/PySide
- wxPython
- DearPyGui
- EasyGUI

Для створення мобільних додатків:

- Kivy
- Toga

Для створення веб-застосунків:

- Django
- Flask
- FastAPI
- Pylons
- Bottle
- CherryPy
- TurboGears

- Nagare

Для автоматизації процесів:

- Selenium (для тестування веб-додатків)
- Flask
- FastAPI
- Pylons
- Bottle
- CherryPy
- TurboGears
- Nagare
- robotframework
- pywinauto
- Lettuce
- Behave
- Requests

Для роботи з різними типами файлів:

- OpenPyXL (Excel)
- lxml (XML)
- ReportLab/borb (PDF)
- pdfrw / PyPDF2 (PDF)
- Pandas (CSV та Excel)

Для машинного навчання, штучного інтелекту, Data Science:

- Pandas
- SciPy
- PyTorch
- Matplotlib
- Theano
- Tensorflow
- OpenCV
- Scikit-Learn
- Keras
- NumPy

Для візуалізації:

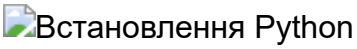
- Matplotlib
- Seaborn

- Plotly
- Bokeh
- Altair
- HoloViews

Встановлення та перша програма на Windows

Встановлення

Для створення програм на Python нам знадобиться інтерпретатор. Для його встановлення перейдемо на сторінку https://www.python.org/downloads/ та знайдемо посилання на завантаження останньої версії мови:


Встановлення Python

Після натискання на кнопку буде завантажено відповідною поточною ОС установщик Python. Слід враховувати, що Windows 7 і попередні версії не підтримуються.

На ОС Windows під час запуску інсталятора запускає вікно майстра установки:


Установка дистрибутива Python 3.12 на Windows

Тут ми можемо задати шлях, яким встановлюватиметься інтерпретатор. Залишимо його за умовчанням, тобто C:\Users[ім'я_користувача]\AppData\Local\Programs\Python\Python312\ .

Крім того, в самому низу відзначимо прапорець "Add Python 3.12 to PATH", щоб додати шлях до інтерпретатора у змінні середовища.

Після цього ми можемо перевірити інсталяцію Python та його версію, запустивши в командному рядку/треміналі команду python --version

```
C:\Users\Nikita>python --version
Python 3.12.1
C:\Users\Nikita>
```

Запуск інтерпретатора

Після встановлення інтерпретатора, як було описано в попередній темі, ми можемо почати створювати програми на Python. Отже, створимо першу просту програму.

Якщо при установці не було змінено адресу, то на Windows Python за замовчуванням встановлюється шляхом

C:\Users[ім'я_користувача]\AppData\Local\Programs\Python\Python[номер_версії]\ і представляє файл під назвою python.exe .


Інтерпретатор Python

Запустимо інтерпретатор і введемо до нього наступний рядок:

```
print("hello world")
```

І консоль виведе рядок "hello world":

```
python 3.12.0 (tags/v3.12.0:0fb18b0, Oct 2 2023, 13:03:39) [MSC
v.1935 64 bit (AMD64)] on win32
Тип "help", "copyright", "кредити" або "license" для більше
інформації.
>>> print("hello world")
hello world
>>>
```

Для цієї програми використовувалася функція print() , яка виводить рядок на консоль.

Створення файлу програми

Насправді програми зазвичай визначаються у зовнішніх файлах-скриптах і потім передаються інтерпретатору на виконання. Тому створимо файл програми. Для цього на диску C або в іншому місці файлової системи визначимо для скриптів папку python . А в цій папці створимо новий текстовий файл, який назвемо hello.py . За замовчуванням файли з кодом Python, як правило, мають розширення py .


Створення скрипта мовою Python

Відкриємо цей файл у будь-якому текстовому редакторі і додамо до нього наступний код:

```
name = input("Введіть ім'я: ")
print("Привіт", name)
```


Python у Visual Studio Code

Скрипт складається із двох рядків. Перший рядок за допомогою функції input() чекає на введення користувача свого імені. Введене ім'я потім потрапляє до змінної `name` .

Другий рядок за допомогою функції `print()` виводить вітання разом із введеним ім'ям.

Тепер запустимо командний рядок/термінал і за допомогою команди cd перейдемо до папки, де знаходиться файл із вихідним кодом hello.py (наприклад, у моєму випадку це папка C:\python).

```
cd c:\python
```

Далі спочатку введемо повний шлях до інтерпретатора, потім повний шлях до файлу скрипта. Наприклад, у моєму випадку в консоль треба буде вести:

```
C:\Users\Nikita\AppData\Local\Programs\Python\Python312\python.exe
hello.py
```

Але якщо при установці була вказана опція "Add Python 3.12 to PATH" , тобто шлях до інтерпретатора Python був доданий до змінних середовищ, то замість повного шляху до інтерпретатора можна просто написати python:

```
python hello.py
```

Або навіть можна скоротити:

```
py hello.py
```

Варіанти з обома способами запуску:

```
Microsoft Windows [Version 10.0.22621.2361]
(c) Корпорація Майкрософт (Microsoft Corporation). Усі права
захищені.
C:\Users\Nikita>cd c:\python
c:\python>C:\Users\Nikita\AppData\Local\Programs\Python\Python312\pyt
hon.exe hello.py
Введіть ім'я: Nikita
Привіт, Nikita
c:\python>python hello.py
Введіть назву: Tom
Привіт, Tom
c:\python>py hello.py
Введіть ім'я: Bob
Привіт, Bob
c:\python>
```

У результаті програма виведе запрошення до введення імені, та був привітання.

Встановлення та перша програма на Mac OS

Встановлення

Для створення програм на Python нам знадобиться інтерпретатор. Для його встановлення перейдемо на сторінку https://www.python.org/downloads/ та знайдемо посилання на завантаження останньої версії мови:

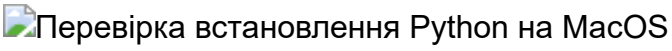

Встановлення Python на MacOS

Якщо поточна ОС - Mac OS, за адресою https://www.python.org/downloads/ буде запропоновано завантажити графічний інсталятор для MacOS. Завантажимо, запустимо його та виконаємо покрокову установку:


Встановлення Python на MacOS

Для звернення до інтерпретатора Python на MacOS застосовується команда python3 . Наприклад, після встановлення інтерпретатора перевіримо його версію командою

```
python3 --version
```

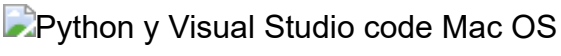

Перевірка встановлення Python на MacOS

Перша програма

Спочатку визначимо десь на жорсткому диску для скриптів папку python . А в цій папці створимо новий текстовий файл, який назвемо hello.py . За замовчуванням файли з кодом Python, як правило, мають розширення py .

Відкриємо цей файл у будь-якому текстовому редакторі і додамо до нього наступний код:

```
name = input("Your name: ")
print("Hello, ", name)
```


Python у Visual Studio code Mac OS

Скрипт складається із двох рядків. Перший рядок за допомогою функції input() чекає на введення користувача свого імені. Введене ім'я потім потрапляє до змінної `name`.

Другий рядок за допомогою функції `print()` виводить вітання разом із введеним ім'ям.

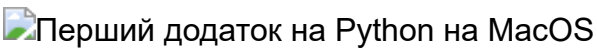
Тепер запустимо командний рядок/термінал і за допомогою команди cd перейдемо до папки, де знаходиться файл із вихідним кодом hello.py (наприклад, у моєму випадку це папка "/Users/Nikita/Documents/python").

```
cd /Users/Nikita/Documents/python
```

Далі для виконання скрипту hello.py передамо його інтерпретатору python:

```
python3 hello.py
```

У результаті програма виведе запрошення до введення імені, та був привітання.

 Перший додаток на Python на MacOS

Встановлення та перша програма на Linux

Встановлення

Для створення програм на Python нам знадобиться інтерпретатор. Варто зазначити, що в деяких дистрибутивах Linux (наприклад, в Ubuntu) Python може бути встановлений за умовчанням. Для перевірки версії Python у терміналі потрібно виконати наступну команду

```
python3 --version
```

 Версія інтерпретатора Python у Linux

Якщо Python встановлений, вона відобразить версію інтерпретатора.

Однак навіть якщо Python встановлений, його версія може бути не останньою. Для встановлення останньої доступної версії Python виконаємо таку команду:

```
sudo apt-get update && sudo apt-get install python3
```

Якщо потрібно встановити не останню доступну, а певну версію, то вказується також підверсія Python. Наприклад, встановлення версії Python 3.10:

```
sudo apt-get install python3.10
```

Відповідно, встановлення версії Python 3.11:

```
sudo apt-get install python3.11
```

Запуск інтерпретатора

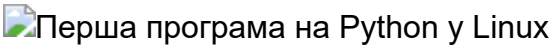
Після встановлення інтерпретатора ми можемо почати створювати програми на Python. Отже, створимо першу просту програму. Для цього введемо в терміналі

```
python3
```

В результаті запускається інтерпретатор Python. Введемо до нього наступний рядок:

```
print("hello METANIT.COM")
```

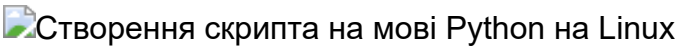
І консоль виведе рядок "hello METANIT.COM":

 Перша програма на Python у Linux

Для цієї програми використовувалася функція print() , яка виводить рядок на консоль.

Створення файлу програми

Насправді програми зазвичай визначаються у зовнішніх файлах-скриптах і потім передаються інтерпретатору на виконання. Тому створимо файл програми. Для цього визначимо для скриптів папку python . А в цій папці створимо новий текстовий файл, який назвемо hello.py . За замовчуванням файли з кодом Python, як правило, мають розширення py .

 Створення скрипта на мові Python на Linux

Відкриємо цей файл у будь-якому текстовому редакторі і додамо до нього наступний код:

```
name = input("Введіть ім'я: ")
print("Привіт", name)
```

 Python у Visual Studio Code

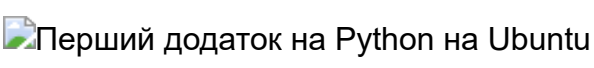
Скрипт складається із двох рядків. Перший рядок за допомогою функції input() чекає на введення користувача свого імені. Введене ім'я потім потрапляє до змінної `name` .

Другий рядок за допомогою функції `print()` виводить вітання разом із введеним ім'ям.

Тепер запустимо термінал та за допомогою команди cd перейдемо до папки, де знаходиться файл з вихідним кодом hello.py (наприклад, у моєму випадку це папка `metanit/python`
в каталозі поточного користувача). І потім виконаємо код `hello.py` за допомогою наступної команди

```
python3 hello.py
```

У результаті програма виведе запрошення до введення імені, та був привітання.

 Перший додаток на Python на Ubuntu

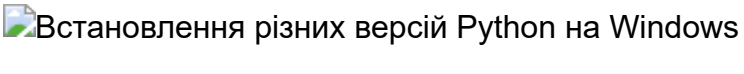
Керування версіями Python на Windows, MacOS та Linux
На одній робочій машині одночасно можна встановити кілька версій Python. Це буває корисно, коли йде робота з деякими зовнішніми бібліотеками, які підтримують різні версії python, або з якихось інших причин нам треба використовувати кілька різних версій. Наприклад, на момент написання статті останньою та актуальною є версія Python 3.11 . Але, скажімо, потрібно також встановити версію 3.10 , як у цьому випадку керувати окремими версіями Python?

Windows

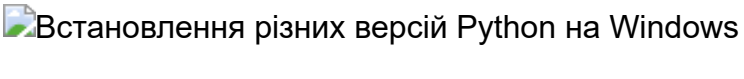
На сторінці завантажень https://www.python.org/downloads/ ми можемо знайти посилання на потрібну версію:

 Управління кількома версіями Python

І також завантажити її та встановити:

 Встановлення різних версій Python на Windows

Щоб при використанні інтерпретатора Python не прописувати до нього весь шлях, додамо при встановленні в змінні середовища. Але тут треба враховувати, що в змінних середовищах може міститися кілька шляхів до різних інтерпретаторів Python:

 Встановлення різних версій Python на Windows у змінні середовища

Та версія Python, яка знаходиться вище, буде стандартною версією. За допомогою кнопки "Вгору" можна потрібну нам версію перемістити на початок, зробивши версією за замовчуванням. Наприклад, у моєму випадку це версія 3.11. Відповідно, якщо я введу в терміналі команду

```
python --version
```

або

```
py --version
```

то консоль відобразить версію 3.11:

```
C:\python>python --version
Python 3.11.0
```

Для звернення до версії 3.10 (та для всіх інших версій) необхідно використовувати вказувати номер версії:

```
C:\python>py -3.10 --version
Python 3.10.9
```

наприклад, виконання скрипту `hello.py` за допомогою версії 3.10:

```
py -3.10 hello.py
```

Так само можна викликати й інші версії Python.

MacOS

На MacOS можна встановити різні версії, наприклад, завантаживши з офіційного сайту пакет інсталятора для певної версії.

Для звернення до певної версії Python на MacOS вказуємо явно підверсію у форматі `python3.[номер_подверсии]` . Наприклад, я маю версію Python 3.10. Перевіримо її версію:

```
python3.10 --version
```

Аналогічно звернення до версії `python3.9` (за умови, якщо вона встановлена)

```
python3.9 --version
```

Наприклад виконання скрипту `hello.py` за допомогою версії python 3.10:

```
python3.10 hello.py
```

Linux

На Linux можна встановити одночасно кілька версій Python. Наприклад, встановлення версій 3.10 та 3.11:

```
sudo apt-get install python3.10
sudo apt-get install python3.11
```

Однією з версій є стандартна версія. І для звернення до неї достатньо прописати python3 , наприклад, перевіримо версію за замовчуванням:

```
python3 --version
```

Для звернення до інших версій треба вказувати підверсію:

```
python3.10 --version
python3.11 --version
```

Наприклад, виконання скрипту `hello` за допомогою версії Python 3.10:

```
python3.10 hello.py
```

Але може скластися ситуація, коли нам треба змінити стандартну версію. У цьому випадку використовується команда update-alternatives для зв'язування певної версії Python з командою python3. Наприклад, ми хочемо встановити як версію за промовчанням Python 3.11. У цьому випадку послідовно виконаємо такі команди:

```
sudo update-alternatives --install /usr/bin/python3 python3
/usr/bin/python3.10 1
sudo update-alternatives --install /usr/bin/python3 python3
/usr/bin/python3.11 2
```

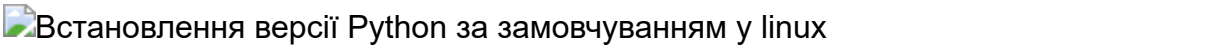
Числа праворуч вказують на пріоритет/стан. Так, для версії 3.11 вказано більший пріоритет, тому при зверненні до використання `python3` буде використовуватися саме версія 3.11 (у моєму випадку це Python 3.11.0rc1)


Управління версіями Python в linux

За допомогою команди

```
sudo update-alternatives --config python3
```

можна змінити стандартну версію


Встановлення версії Python за замовчуванням у linux

Перша програма в PyCharm
Минулої теми було описано створення найпростішого скрипта мовою Python. Для створення скрипта використовувався текстовий редактор. У моєму випадку це був Notepad. Але є інший спосіб створення програм, який представляє використання різних інтегрованих середовищ розробки або IDE.

IDE надають нам текстовий редактор для набору коду, але на відміну від стандартних текстових редакторів, IDE також забезпечує повноцінне підсвічування синтаксису, автодоповнення або інтелектуальну підказку коду, можливість відразу виконати створений скрипт, а також багато іншого.

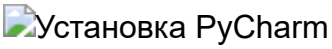
Для Python можна використовувати різні середовища розробки, але однією з найпопулярніших серед них є середовище PyCharm , створене компанією JetBrains. Це середовище динамічно розвивається, постійно оновлюється і доступне найбільш поширених операційних систем - Windows, MacOS, Linux.

Щоправда, вона має одне важливе обмеження. А саме вона доступна у двох основних варіантах: платний випуск Professional та безкоштовний Community. Багато базових можливостей доступні і в безкоштовному випуску Community. У той же час ряд можливостей, наприклад, веб-розробка доступні тільки в платному Professional.

У нашому випадку скористаємось безкоштовним випуском Community. Для цього перейдемо на
сторінку завантаження та завантажимо інсталяційний файл PyCharm Community.


IDE PyCharm

Після завантаження виконаємо його встановлення.

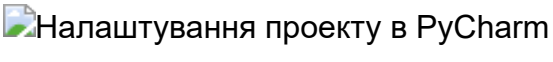

Установка PyCharm

Після завершення встановлення запустимо програму. При першому запуску відкривається початкове вікно:


Перша програма в PyCharm

Створимо проект і для цього оберемо пункт New Project .

Далі нам відкриється вікно для налаштування проекту. У полі Location необхідно вказати шлях до проекту. У моєму випадку проект буде розміщено в папці HelloApp. Власне назва папки і буде назвою проекту.


Налаштування проекту в PyCharm

Крім шляху до проекту, всі інші налаштування залишимо за замовчуванням і натиснемо на кнопку Create для створення проекту.

Після цього буде створено порожній проект:

 Перший проект у PyCharm

У центрі середовища буде відкрито файл main.py з деяким вмістом за промовчанням.

Тепер створимо найпростішу програму. Для цього змінимо код файлу main.py так:

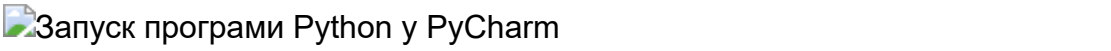
```
name = input("Введіть ваше ім'я: ")
print("Привіт", name)
```

Для запуску скрипту натиснемо на зелену стрілку в панелі інструментів програми:

 Запуск програми у PyCharm

Також для запуску можна перейти в меню Run і там натиснути на підпункт Run 'main')

Після цього внизу IDE відобразиться вікно виведення, де потрібно буде ввести ім'я і де після цього буде виведено привітання:

 Запуск програми Python у PyCharm

Python у Visual Studio
Одним із середовищ розробки, що дозволяє працювати з Python, є Visual Studio. Перевагою даної IDE порівняно, скажімо, з PyCharm, слід зазначити насамперед те, що в її безкоштовній редакції Visual Studio Community безкоштовно доступні ряд функцій і можливостей, які в тому ж PyCharm доступні лише платній версії Professional Edition. Наприклад, це веб-розробка, у тому числі за допомогою різних фреймворків. У той же час засоби для розробки на Python у Visual Studo доступні поки що тільки у версії для Windows.

Отже, завантажимо інсталяційний файл Visual Studio Community за посиланням https://visualstudio.microsoft.com/ru/vs/community/ . Після запуску інсталяційного файлу виберемо серед опцій пункт Розробка на Python :

 Встановлення Python у Visual Studio

Після інсталяції Visual Studio запустимо її і у вікні програми оберемо Create a new project :

 Створення проекту для Python у Visual Studio

Далі у вікні створення нового проекту виберемо шаблон Python Application :

На наступному вікні вкажемо назву та шлях до проекту. Наприклад, у моєму випадку проект називатиметься "HelloApp":

Натисніть кнопку Create, і Visual Studio створить новий проект:

Праворуч у вікні Solution Explorer (Обозреватель рішень) можна побачити структуру проекту. За замовчуванням ми можемо побачити такі елементи:

- Python Environments : тут можна побачити всі використовувані середовища, зокрема тут можна версію Python, яка використовується.
- References : цей вузол містить всі зовнішні залежності, які використовуються поточним проектом
- Search Paths : цей вузол дозволяє вказати шляхи пошуку для модулів Python
- HelloApp.py : власне файл Python з вихідним кодом

За промовчанням у Visual Studio вже відкрито файл HelloApp.py, але він поки що порожній. Додамо до нього наступний рядок:

```
print("Hello Python from Visual Studio!")
```

І потім на панелі інструментів натиснемо на зелену стрілочку для запуску:

В результаті запуску відобразиться консоль, яка виведе потрібний рядок:

Розділ 2. Основи Python

Введення у написання програм

Програма Python складається з набору інструкцій. Кожна інструкція міститься на новий рядок. Наприклад:

```
print(2 + 3)
print("Hello")
```

Велику роль у Python відіграють відступи. Неправильно поставлений відступ фактично є помилкою. Наприклад, у наступному випадку ми отримаємо помилку, хоча код буде практично аналогічний наведеному вище:

```
print(2 + 3)
    print("Hello")
```

Тому варто поміщати нові вказівки спочатку рядка. У цьому одна з важливих відмінностей пайтон від інших мов програмування, як C# або Java.

Однак, варто враховувати, що деякі конструкції мови можуть складатися з декількох рядків. Наприклад, умовна конструкція if :

```
if 1 < 2:
    print("Hello")
```

У разі якщо 1 менше 2, то виводиться рядок " Hello " . І тут має бути відступ, оскільки інструкція print("Hello") використовується не як така, бо як частина умовної конструкції if. Причому відступ, згідно з посібником з оформлення коду , бажано робити з такої кількості прогалин, яка кратна 4 (тобто 4, 8, 16 і т.д.) Хоча якщо відступів буде не 4, а 5, то програма також працюватиме.

Таких конструкцій не так багато, тому особливої плутанини щодо де треба, а де не треба ставити прогалини, не повинно виникнути.

Реєстрозалежність

Python - регістрозалежна мова, тому вирази `print` та `Print` або `PRINT` представляють різні вирази. І якщо замість методу print для виведення на консоль ми спробуємо використати метод Print:

```
Print("Hello World")
```

то в нас нічого не вийде.

Коментарі

Для позначки, що робить та чи інша ділянка коду, застосовуються коментарі. При трансляції та виконанні програми інтерпретатор ігнорує коментарі, тому вони не впливають на роботу програми. Коментарі в Python бувають блокові та малі.

Рядкові коментарі передуються знаком решітки -# . Вони можуть розташовуватися на окремому рядку:

```
# Виведення на консоль
# повідомлення Hello World
```

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

Будь-який набір символів після знак# представляє коментар. Тобто у прикладі вище перші два рядки коду є коментарями.

Також вони можуть розташовуватися на тому ж рядку, що й інструкції мови після інструкцій, що виконуються:

```
print("Hello World")# Виведення повідомлення на консоль
```

У блокових коментарях до і після тексту коментаря ставляться три одинарні лапки: `'''текст комментария'''`. Наприклад:

```
'''
    Виведення на консоль
    повідомлення Hello World
'''
print("Hello World")
```

Основні функції

Python надає низку вбудованих функцій. Деякі їх використовуються дуже часто, особливо у початкових етапах вивчення мови, тому розглянемо їх.

Основною функцією виведення інформації на консоль є функція print() . Як аргумент у цю функцію передається рядок, який ми хочемо вивести:

```
print("Hello Python")
```

Якщо нам необхідно вивести кілька значень на консоль, ми можемо передати їх у функцію print через кому:

```
print("Full name:", "Tom", "Smith")
```

У результаті всі передані значення склеяться через прогалини в один рядок:

```
Full name: Tom Smith
```

Якщо функція print відповідає за виведення, функція input відповідає за введення інформації. Як необов'язковий параметр ця функція приймає запрошення до введення та повертає введений рядок, який ми можемо зберегти в змінний:

```
name = input("Введіть ім'я: ")
print("Привіт", name)
```

Консольний висновок:

```
Введіть ім'я: Нікіта
Привіт Нікіта
```

Змінні та типи даних

Змінні

Змінні призначені для зберігання даних. Назва змінної в Python повинна починатися з алфавітного символу або символу підкреслення і може містити алфавітно-цифрові символи і знак підкреслення. Крім того, назва змінної не повинна співпадати з назвою ключових слів мови Python. Ключових слів не так багато, їх легко запам'ятати:

```
False await else import pass
None break except in raise
True class finally is return
and continue for lambda try
as def from nonlocal while
assert del global not with
async elif if або yield
```

Наприклад, створимо змінну:

```
name = "Tom"
```

Тут визначено змінну `name`, яка зберігає рядок "Tom".

У пайтоні застосовується два типи найменування змінних: camel case та underscore notation .

Camel case має на увазі, що кожне нове підслівне в найменуванні змінної починається з великої літери. Наприклад:

```
userName = "Tom"
```

Underscore notation передбачає, що підслів в найменуванні змінної поділяються знаком підкреслення. Наприклад:

```
user_name = "Tom"
```

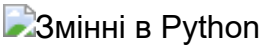
І також треба враховувати регістрозалежність, тому змінні `name` і `Name` представлятимуть різні об'єкти.

```
дві різні змінні
name = "Tom"
Name = "Tom"
```

Визначивши змінну ми можемо використовувати в програмі. Наприклад, спробувати вивести її на консоль за допомогою вбудованої функції print :

```
name = "Tom"# Визначення змінної name
print(name)# виведення значення змінної name на консоль
```

Наприклад, визначення та застосування змінної в середовищі PyCharm:

 Змінні в Python

Відмінною рисою змінної є те, що ми можемо змінювати її значення протягом роботи програми:

```
name = "Tom"# змінної name дорівнює "Tom"
print(name)# виводить: Tom
name = "Bob"# змінюємо значення на "Bob"
print(name)# виводить: Bob
```

Типи даних

Змінна зберігає дані одного з типів даних. У Python існує безліч різних типів даних. В даному випадку розглянемо тільки базові типи:
bool , int , float , complex і str .

Логічні значення

Тип bool представляє два логічні значення: True (вірно, істина) або False (невірно, брехня). Значення
True служить у тому, щоб показати, що щось істинно. Тоді як значення False , навпаки, показує, що щось хибне. Приклад змінних даного типу:

```
isMarried = False
print(isMarried)# False
isAlive = True
print(isAlive)# True
```

Цілі числа

Тип int є цілим числом, наприклад, 1, 4, 8, 50. Приклад

```
age = 21
print("Вік:", age)# Вік: 21
count = 15
print("Кількість:", count)# Кількість: 15
```

За замовчуванням стандартні числа розцінюються як числа у десятковій системі. Але Python також підтримує числа у двійковій, вісімковій та шістнадцятковій системах.

Для вказівки, що число є двійковою системою, перед числом ставиться префікс 0b :

```
a = 0b11
b = 0b1011
c = 0b100001
print(a)# 3 у десятковій системі
print(b)# 11 у десятковій системі
print(c)# 33 у десятковій системі
```

Для вказівки, що число представляє вісімкову систему, перед числом ставиться префікс 0o :

```
a = 0o7
b = 0o11
c = 0o17
print(a)# 7 у десятковій системі
print(b)# 9 у десятковій системі
print(c)# 15 у десятковій системі
```

Для вказівки, що число є шістнадцятковою системою, перед числом ставиться префікс 0x :

```
a = 0x0A
b = 0xFF
c = 0xA1
print(a)# 10 у десятковій системі
print(b)# 255 у десятковій системі
print(c)# 161 у десятковій системі
```

У будь-якій системі ми не передали число в функцію print для виведення на консоль, воно за умовчанням буде виводитися в десятковій системі.

Дробові числа

Тип float є число з плаваючою точкою, наприклад, 1.2 або 34.76. У якості роздільника цілої та дробової частин використовується крапка.

```
height = 1.68
pi = 3.14
weight = 68.
print(height)# 1.68
print(pi)# 3.14
print(weight)# 68.0
```

Число з плаваючою точкою можна визначати в експоненційному записі:

```
x = 3.9e3
print(x)# 3900.0
x = 3.9e-3
print(x)# 0.0039
```

Число float може мати лише 18 значущих символів. Так, у цьому випадку використовуються лише два символи – 3.9. І якщо число занадто велике чи занадто мало, ми можемо записувати число у подібній нотації, використовуючи експоненту. Число після експоненти вказує ступінь числа 10, яке треба помножити основне число - 3.9.

Комплексні числа

Тип complex представляє комплексні числа у форматі - після уявної частини вказується суфікс j `вещественная_часть+мнимая_частьj`

```
complexNumber = 1+2j
print(complexNumber)# (1+2j)
```

Рядки

Тип str представляє рядки. Рядок представляє послідовність символів, укладену в одинарні або подвійні лапки, наприклад "hello" та "hello". У Python 3.x рядки представляють набір символів у кодуванні Unicode

```
message = "Hello World!"
print(message)# Hello World!
name = 'Tom'
print(name)# Tom
```

При цьому, якщо рядок має багато символів, його можна розбити на частини та ці частини розмістити на різних рядках коду. У цьому випадку весь рядок полягає в круглі дужки, а її окремі частини - у лапки:

```
text = ("Laudate omnes gentes laudate"
        "Magnificat in secula")
print(text)
```

Якщо ми хочемо визначити багаторядковий текст, такий текст полягає у потрійні подвійні чи одинарні лапки:

```
'''
Це коментар
'''
text = '''Laudate omnes gentes laudate
Magnificat in secula
Et anima mea laudate
Magnificat in secula
'''
print(text)
```

При використанні потрійних одинарних лапок не варто плутати їх із коментарями: якщо текст у потрійних одинарних лапках присвоюється змінним, то це рядок, а не коментар.

Керуючі послідовності у рядку

Рядок може містити ряд спеціальних символів - послідовностей, що управляють. Деякі з них:

- \ : дозволяє додати всередину рядки слєш
- ' : дозволяє додати всередину рядка одинарну лапку
- " : дозволяє додати всередину рядка подвійну лапку
- \n : здійснює перехід на новий рядок
- \t : додає табуляцію (4 відступи)

Застосуємо кілька послідовностей:

```
text = "Message:\n\"Hello World\""
print(text)
```

Консольний висновок програми:

```
Message:
"Hello World"
```

Хоча подібні послідовності можуть допомогти в деяких справах, наприклад, помістити в рядок лапку, зробити табуляцію, перенесення на інший рядок. Але вони також можуть заважати. Наприклад:

```
path = "C:\python\name.txt"
print(path)
```

Тут змінна path містить певний шлях до файлу. Однак усередині рядка зустрічаються символи "\n", які будуть інтерпретовані як послідовність, що управляє. Так, ми отримаємо наступний консольний висновок:

```
C:\python
ame.txt
```

Щоб уникнути подібної ситуації, перед рядком ставиться символ r

```
path = r"C:\python\name.txt"
print(path)
```

Вставка значень у рядок

Python дозволяє встроювати рядок значення інших змінних. Для цього всередині рядка змінні розміщуються у фігурних дужках {}, а перед усім рядком ставиться символ f :

```
userName = "Tom"
userAge = 37
user = f"name: {userName} age: {userAge}"
print(user)# name: Tom age: 37
```

У цьому випадку на місце `{userName}` вставлятиметься значення змінної userName. Аналогічно замість `{userAge}`
буде вставлятися значення змінної userAge.

Динамічна типізація

Python є мовою з динамічною типізацією. А це означає, що змінна не прив'язана до певного типу.

Тип змінної визначається виходячи із значення, яке їй надано. Так, при присвоєнні рядка в подвійних чи одинарних лапках змінна має тип str . При присвоєнні цілого числа Python автоматично визначає тип змінної як int . Щоб визначити змінну як об'єкт float, їй надається дробове число, в якому роздільником цілої та дробової частини є точка.

При цьому в процесі роботи програми ми можемо змінити тип змінної, надавши їй значення іншого типу:

```
userId = "abc"# тип str
print(userId)
userId = 234# тип int
print(userId)
```

За допомогою вбудованої функції type() динамічно можна дізнатися про поточний тип змінної:

```
userId = "abc"# тип str
print(type(userId))# <class 'str'>
userId = 234# тип int
print(type(userId))# <class 'int'>
```

Консольне введення та виведення

Виведення на консоль

Для виведення інформації на консоль призначена вбудована функція print() . При виклику цієї функції їй у дужках передається значення:

```
print("Hello METANIT.COM")
```

Цей код виведе нам на консоль рядок "Hello METANIT.COM".

Відмінною особливістю цієї функції є те, що за умовчанням вона виводить значення на окремому рядку. Наприклад:

```
print("Hello World")
print("Hello METANIT.COM")
print("Hello Python")
```

Тут три дзвінки функції `print()` виводять деяке повідомлення. Причому при виведенні на консоль кожне повідомлення розміщуватиметься на окремому рядку:

```
Hello World
Hello METANIT.COM
Hello Python
```

Така поведінка не завжди зручна. Наприклад, ми хочемо, щоб усі значення виводилися на одному рядку. Для цього нам потрібно настроїти поведінку функції за допомогою параметра end . Цей параметр визначає символи, які додаються в кінці до рядка і . При застосуванні параметра end виклик функції `print()` виглядає так:

```
print(значення, end = кінцеві_символи)
```

За промовчанням `end` дорівнює символу `"\n"`, який задає переклад наступного рядка. Власне тому функція print за замовчуванням виводить значення, що їй передається, на окремому рядку.

Тепер визначимо, щоб функція не робила переведення на наступний рядок, а виводила значення на тому ж рядку:

```
print("Hello World", end=" ")
print("Hello METANIT.COM", end=" ")
print("Hello Python")
```

Тобто тепер значення, що виводяться, будуть розділятися пробілом:

```
Hello World Hello METANIT.COM Hello Python
```

Причому це може бути не один символ, а набір символів:

```
print("Hello World", end=" and ")
print("Hello METANIT.COM", end=" and ")
print("Hello Python")
```

В даному випадку повідомлення, що виводяться, будуть відокремлюватися символами `" and "`:

```
Hello World and Hello METANIT.COM and Hello Python
```

Консольне введення

Поряд із виведенням на консоль ми можемо отримувати введення користувача з консолі, отримувати дані, що вводяться. Для цього Python визначена функція

input() . У цю функцію надсилається запрошення до введення. А результат введення ми можемо зберегти змінну. Наприклад, визначимо код для введення користувачем імені:

```
name = input("Введіть своє ім'я: ")
print(f"Ваше ім'я: {name}")
```

У цьому випадку функцію `input()` передається запрошення до введення у вигляді рядка "Введіть своє ім'я: ". Результат функції - результат введення користувача передається до змінної `name` . Потім ми можемо вивести значення цієї змінної консоль за допомогою функції `print()` . Приклад роботи коду:

```
Введіть своє ім'я: Nikita
Ваше ім'я: Nikita
```

Ще приклад із введенням кількох значень:

```
name = input("Your name: ")
age = input("Your age: ")
print(f"Name: {name} Age: {age}")
```

Приклад роботи програми:

```
Your name: Tom
Your age: 37
Name: Tom Age: 37
```

Варто враховувати, що усі введені значення розглядаються як значення типу str , тобто рядки. І навіть якщо ми вводимо число, як у другому випадку у коді вище, то Python все одно розглядатиме введене значення як рядок, а не як число.

Арифметичні операції з числами
Python підтримує всі поширені арифметичні операції:

- Додавання двох чисел:

```
print(6 + 2)# 8
```

- Віднімання двох чисел:

```
print(6 - 2)# 4
```

- *

  Розмноження двох чисел:

  ```
  print(6 * 2)# 12
  ```

- /

  Розподіл двох чисел:

  ```
  print(6 / 2)# 3.0
  ```

- //

  Цілочисельне розподіл двох чисел:

  ```
  print(7 / 2)# 3.5
  print(7 // 2)# 3
  ```

  Ця операція повертає цілий результат поділу, відкидаючи дробову частину

- **

  Зведення у ступінь:

  ```
  print(6 ** 2)# Зводимо число 6 до ступеня 2. Результат - 36
  ```

- %

  Отримання залишку від розподілу:

  ```
  print(7 % 2)# Отримання залишку від розподілу числа 7 на 2.
  Результат - 1
  ```

  В даному випадку найближче число до 7, яке ділиться на 2 без залишку, це 6. Тому залишок від поділу дорівнює 7 - 6 = 1

При послідовному використанні кількох арифметичних операцій їх виконання здійснюється відповідно до пріоритету. Спочатку виконуються операції з великим пріоритетом. Пріоритети операцій у порядку зменшення наведені в наступній таблиці.

Операції	**Напрям**
**	Справа наліво

Операції	Напрям
* // // %	Зліва направо
+ -	Зліва направо

Нехай у нас виконується такий вираз:

```
number = 3 + 4 * 5 ** 2 + 7
print(number)# 110
```

Тут на початку виконується зведення у ступінь (5**2) як операція з великим пріоритетом, далі результат множиться на 4 (25*4), потім відбувається додавання (3+100) і далі знову йде додавання (103+7).

Щоб перевизначити порядок операцій, можна використовувати дужки:

```
number = (3 + 4) * (5 ** 2 + 7)
print(number)# 224
```

Слід зазначити, що у арифметичних операціях можуть брати участь як цілі, і дробові числа. Якщо однієї операції бере участь ціле число (int) і з плаваючою точкою (float), то ціле число наводиться до типу float.

Арифметичні операції із присвоєнням

Ряд спеціальних операцій дозволяють використовувати результат операції першому операнду:

- +=

  Присвоєння результату додавання
- -=

  Присвоєння результату віднімання
- *=

  Присвоєння результату множення
- /=

  Присвоєння результату від розподілу
- //=

  Присвоєння результату цілісного поділу
- **=

  Присвоєння ступеня числа

- %=

  Присвоєння залишку від розподілу

Приклади операцій:

```
number = 10
number += 5
print(number)# 15
number -= 3
print(number)# 12
number *= 4
print(number)# 48
```

Округлення та функція round

При операціях з числами типу float треба враховувати, що результати операцій з ними може бути не зовсім точним. Наприклад:

```
first_number = 2.0001
second_number = 5
third_number = first_number/second_number
print(third_number)# 0.40002000000000004
```

В даному випадку ми очікуємо отримати число 0.40002, проте в кінці через ряд нулів з'являється ще якась четвірка. Або ще один вираз:

```
print(2.0001 + 0.1)# 2.1001000000000003
```

У разі вище заокруглення результату ми можемо використовувати вбудовану функцію round() :

```
first_number = 2.0001
second_number = 0.1
third_number = first_number + second_number
print(round(third_number))# 2
```

У функцію `round()` передається число, яке треба округлити. Якщо функцію передається одне число, як у прикладі вище, воно округляється до цілого.

Функція `round()` також може приймати друге число, яке вказує, скільки знаків після коми має містити число, що отримується:

```
first_number = 2.0001
second_number = 0.1
third_number = first_number + second_number
print(round(third_number, 4))# 2.1001
```

У разі число `third_number` округляється до 4 знаків після коми.

Якщо в функцію передається лише одне значення - тільки округлене число, воно округляється то найближчого цілого

Приклади заокруглень:

```
округлення до цілого числа
print(round(2.49))# 2 - округлення до найближчого цілого 2
print(round(2.51))# 3
```

Однак якщо округлена частина дорівнює однаково віддалена від двох цілих чисел, то округлення йде до найближчого парного:

```
print(round(2.5))# 2 - найближче парне
print(round(3.5))# 4 - найближче парне
```

Округлення проводиться до найближчого кратного 10 в ступені мінус округлена частина:

```
округлення до двох знаків після коми
print(round(2.554, 2))# 2.55
print(round(2.5551, 2))# 2.56
print(round(2.554999, 2))# 2.55
print(round(2.499, 2))# 2.5
```

Однак слід враховувати, що функція `round()` не є ідеальним інструментом. Наприклад, вище за округлення до цілих чисел застосовується правило, згідно з яким, якщо округлена частина однаково віддалена від двох значень, округлення проводиться до найближчого парного значення. У Python у зв'язку з тим, що десяткова частина числа не може бути точно представлена у вигляді числа float, це може призводити до деяких не зовсім очікуваних результатів. Наприклад:

```
округлення до двох знаків після коми
print(round(2.545, 2))# 2.54
print(round(2.555, 2))# 2.56 - округлення до парного
print(round(2.565, 2))# 2.56
print(round(2.575, 2))# 2.58
print(round(2.655, 2))# 2.65 - округлення не до парного
```

```
print(round(2.665, 2))# 2.67
print(round(2.675, 2))# 2.67
```

Подібно до проблеми можна почитати до документації .

Додаткові матеріали

- Запитання для самоперевірки
- Вправи для самоперевірки

Умовні вирази
Ряд операцій представляють умовні висловлювання. Всі ці операції приймають два операнди і повертають логічне значення, яке Python представляє тип bool . Існує тільки два логічні значення - True (вираз істинно) і False (вираз хибно).

Операції порівняння

Найпростіші умовні вирази становлять операції порівняння, які порівнюють два значення. Python підтримує наступні операції порівняння:

- ==

  Повертає True, якщо обидва операнди рівні. Інакше повертає False.
- !=

  Повертає True, якщо обидва операнди НЕ рівні. Інакше повертає False.
- > (більше ніж)

  Повертає True, якщо перший операнд більший за другий.
- < (менше ніж)

  Повертає True, якщо перший операнд менший за другий.
- >= (більше чи одно)

  Повертає True, якщо перший операнд більше або дорівнює другому.
- <= (менше чи одно)

  Повертає True, якщо перший операнд менший або дорівнює другому.

Приклади операцій порівняння:

```
a = 5
b = 6
result = 5 == 6# зберігаємо результат операції у змінну
print(result)# False - 5 не дорівнює 6
```

```
print(a != b)# True
print(a > b)# False - 5 менше 6
print(a < b)# True
bool1 = True
bool2 = False
print(bool1 == bool2)# False - bool1 не дорівнює bool2
```

Операції порівняння можуть порівнювати різні об'єкти - рядки, числа, логічні значення, проте обидва операнди операції повинні представляти один і той же тип.

Логічні операції

Для створення складових умовних виразів використовуються логічні операції. У Python є такі логічні оператори:

- Оператор and (логічне множення) застосовується до двох операндів:

  ```
  x and y
  ```

  Спочатку оператор and оцінює вираз x, і якщо він дорівнює False, то повертається його значення. Якщо воно дорівнює True, то оцінюється другий операнд – y і повертається значення y.

  ```
  age = 22
  weight = 58
  result = age > 21 and weight == 58
  print(result)# True
  ```

  У разі оператор and порівнює результати двох выражений: `age > 21` `weight == 58`. І якщо обидва ці висловлювання повертають True, то оператор and також повертає True (формально повертається значення останнього операнда).

  Але операндами оператора and необов'язково виступають значення True і False. Це можуть бути будь-які значення. Наприклад:

  ```
  result = 4 and "w"
  print(result)# w, тому що 4 дорівнює True, тому повертається
  значення останнього операнда
  result = 0 and "w"
  print(result)# 0, тому що 0 еквівалентно False
  ```

У даному випадку число 0 і порожній рядок "" розглядаються як False, всі інші числа та непусті рядки еквівалентні True

- or (логічне додавання) також застосовується до двох операндів:

```
x or y
```

Спочатку оператор or оцінює вираз x, і якщо він дорівнює True, то повертається його значення. Якщо воно дорівнює False, то оцінюється другий операнд – y і повертається значення y. Наприклад

```
age = 22
isMarried = False
result = age > 21 or isMarried
print(result)# True, тому що вираз age > 21 дорівнює True
```

Також оператор or може застосовуватися до будь-яких значень. Наприклад:

```
result = 4 or "w"
print(result)# 4, тому що 4 еквівалентно True, тому повертається значення першого операнда
result = 0 or "w"
print(result)# w, тому що 0 еквівалентно False, тому повертається значення останнього операнда
```

- not (логічне заперечення)

Повертає True, якщо вираз дорівнює False

```
age = 22
isMarried = False
print(not age > 21)# False
print(not isMarried)# True
print(not 4)# False
print(not 0)# True
```

Оператор in

Оператор in повертає, `True` якщо в деякому наборі значень є певне значення. Він має таку форму:

```
значення in набор_значень
```

Наприклад, рядок представляє набір символів. І за допомогою оператора `in` ми можемо перевірити, чи є в ній якийсь підрядок:

```
message = "hello world!"
hello = "hello"
print(hello in message)# True - підрядок hello є в рядку "hello
world!"
gold="gold"
print(gold in message)# False - підрядки "gold" немає в рядку "hello
world!"
```

Якщо нам треба навпаки перевірити, чи немає в наборі значень будь-якого значення, ми можемо використовувати модифікацію оператора - not in . Вона повертає `True` , якщо в наборі значень немає певного значення:

```
message = "hello world!"
hello = "hello"
print(hello not in message)# False
gold="gold"
print(gold not in message)# True
```

Додаткові матеріали

- Запитання для самоперевірки

Умовна конструкція if
Умовні конструкції використовують умовні вирази і в залежності від їх значення спрямовують виконання програми по одному із шляхів. Одна з таких конструкцій - це конструкція if . Вона має таке формальне визначення:

```
if логічний_вираз:
    інструкції
[elif логічний вираз:
    інструкції]
[else:
    інструкції]
```

У найпростішому вигляді після ключового слова if йде логічний вираз. І якщо цей логічний вираз повертає True, то виконується наступний блок інструкцій, кожна з яких повинна починатися з нового рядка і повинна мати відступи від початку виразу `if` (відступ бажано робити в 4 пробіли або кількість прогалин, яке кратно 4):

```
language = "english"
if language == "english":
```

```
    print("Hello")
print("End")
```

Оскільки в даному випадку значення змінної мови дорівнює "english", то буде виконуватися блок if, який містить тільки одну інструкцію - `print("Hello")`. У результаті консоль виведе такі рядки:

```
Hello
End
```

Зверніть увагу на останній рядок, який виводить повідомлення "End". Вона не має відступів від початку рядка, тому вона не належить до блоку if і виконуватиметься у будь-якому випадку, навіть якщо вираз у конструкції if поверне False.

Але якби ми поставили відступи, то вона також належала б до конструкції if:

```
language = "english"
if language == "english":
    print("Hello")
    print("End")
```

Блок else

Якщо раптом нам треба визначити альтернативне рішення на той випадок, якщо вираз у if поверне False, то ми можемо використати блок else :

```
language = "російський"
if language == "english":
    print("Hello")
else:
    print("Привіт")
print("End")
```

Якщо вираз `language == "english"` повертає True, виконується блок if, інакше виконується блок else. І оскільки в даному випадку умова `language == "english"` повертає False, то виконуватиметься інструкція із блоку `else`.

Причому інструкції блоку else також повинні мати відступи від початку рядка. Наприклад, у прикладі вище `print("End")`
немає відступу, тому вона входить у блок `else` і виконуватиметься незалежно, чому одно умова `language == "english"`. Тобто консоль нам виведе наступні рядки:

```
Привіт
End
```

Блок else також може мати кілька інструкцій, які повинні мати відступ від початку рядка:

```
language = "російський"
if language == "english":
    print("Hello")
    print("World")
else:
    print("Привіт")
    print("світ")
```

elif

Якщо необхідно ввести кілька альтернативних умов, можна використовувати додаткові блоки elif , після якого йде блок інструкцій.

```
language = "німецький"
if language == "english":
    print("Hello")
    print("World")
elif language == "німецький":
    print("Hallo")
    print("Welt")
else:
    print("Привіт")
    print("світ")
```

Спочатку Python перевіряє вираз `if` . Якщо воно дорівнює True, виконуються інструкції з блоку if. Якщо ця умова повертає False, то Python перевіряє вираз із `elif` .

Якщо вираз після `elif` одно `True` , то виконуються інструкції з блоку `elif` . Але якщо воно одно,
`False` то виконуються інструкції з блоку `else`

За потреби можна визначити декілька блоків `elif` для різних умов. Наприклад:

```
language = "німецький"
if language == "english":
    print("Hello")
elif language == "німецький":
    print("Hallo")
```

```
elif language == "french":
    print("Salut")
else:
    print("Привіт")
```

Вкладені конструкції if

Конструкція if у свою чергу сама може мати вкладені конструкції if:

```
language = "english"
daytime = "morning"
if language == "english":
    print("English")
    if daytime == "morning":
        print("Good morning")
    else:
        print("Good evening")
```

Тут конструкція if містить вкладену конструкцію if/else. Тобто якщо змінна language дорівнює "english", тоді вкладена конструкція if/else додатково перевіряє значення змінної daytime - чи дорівнює вона рядку "morning" чи ні. І в даному випадку ми отримаємо наступний консольний висновок:

```
English
Good morning
```

Варто враховувати, що вкладені вирази if також повинні починатися з відступів, а інструкції у вкладених конструкціях повинні мати відступи. Відступи, розставлені належним чином, можуть змінити логіку програми. Так, попередній приклад не аналогічний наступному:

```
language = "english"
daytime = "morning"
if language == "english":
    print("English")
if daytime == "morning":
    print("Good morning")
else:
     print("Good evening")
```

Подібним чином можна розміщувати вкладені конструкції if/elif/else у блоках elif та else:

```
language = "російський"
daytime = "morning"
if language == "english":
    if daytime == "morning":
        print("Good morning")
    else:
        print("Good evening")
else:
    if daytime == "morning":
        print("Доброго ранку")
    else:
        print("Добрий вечір")
```

Додаткові матеріали

- Запитання для самоперевірки
- Вправи для самоперевірки

Цикли
Цикли дозволяють виконувати певну дію залежно від дотримання певної умови. У мові Python є такі типи циклів:

- while
- for

Цикл while

Цикл while перевіряє істинність деякого умови, і якщо умова істинно, то виконує інструкції циклу. Він має таке формальне визначення:

```
while умовний_вираз:
   інструкції
```

Після ключового слова while вказується умовний вираз, і доки цей вираз повертає значення `True`, виконуватиметься блок інструкцій, що йде далі.

Усі інструкції, які відносяться до циклу while, розташовуються на наступних рядках і повинні мати відступ від початку ключового слова while.

```
number = 1
while number < 5:
    print(f"number = {number}")
    number += 1
print("Робота програми завершена")
```

В даному випадку цикл while буде виконуватися, поки змінна number менше 5.

Сам блок циклу і двох інструкцій:

```
print(f"number = {number}")
number += 1
```

Зверніть увагу, що вони мають відступи від початку оператора while - у разі від початку рядка. Завдяки цьому Python може визначити, що вони належать циклу. У самому циклі спочатку виводиться значення змінної number, а потім їй надається нове значення. .

Також зверніть увагу, що остання інструкція `print("Работа программы завершена")` не має відступів від початку рядка, тому вона не входить до циклу while.

Весь процес циклу можна представити так:

1. Спочатку перевіряється значення змінної number - чи воно менше 5. І оскільки спочатку змінна дорівнює 1, то ця умова повертає `True`, і тому виконуються інструкції циклу

   Інструкції циклу виводять на консоль рядок `number = 1`. І далі значення змінної number збільшується на одиницю - тепер вона дорівнює 2. Одноразове виконання блоку інструкцій циклу називається ітерацією . Тобто таким чином у циклі виконується перша ітерація.
2. Знову перевіряється умова `number < 5`. Воно, як і раніше, дорівнює True, тому що number = 2, тому виконуються інструкції циклу

   Інструкції циклу виводять на консоль рядок `number = 2`. І далі значення змінної number знову збільшується на одиницю - тепер вона дорівнює 3. Таким чином, виконується друга ітерація.
3. Знову перевіряється умова `number < 5`. Воно, як і раніше, дорівнює True, тому що number = 3, тому виконуються інструкції циклу

   Інструкції циклу виводять на консоль рядок `number = 3`. І далі значення змінної number знову збільшується на одиницю - тепер вона дорівнює 4. Тобто виконується третя ітерація.
4. Знову перевіряється умова `number < 5`. Воно, як і раніше, дорівнює True, тому що number = 4, тому виконуються інструкції циклу

   Інструкції циклу виводять на консоль рядок `number = 4`. І далі значення змінної number знову збільшується на одиницю - тепер вона дорівнює 5. Тобто виконується четверта ітерація.
5. І знову перевіряється умова `number < 5`. Але тепер воно дорівнює False , тому що number = 5, тому виконуються вихід із циклу. Усі цикл – завершився.

Далі вже виконуються дії, визначені після циклу. Таким чином, даний цикл проведе чотири проходи або чотири ітерації

У результаті під час виконання коду ми отримаємо наступний консольний висновок:

```
number = 1
number = 2
number = 3
number = 4
Роботу програми завершено
```

Для циклу while також можна визначити додатковий блок else , інструкції якого виконуються, коли умова дорівнює False:

```
number = 1
while number < 5:
    print(f"number = {number}")
    number += 1
else:
    print(f"number = {number}. Робота циклу завершена")
print("Робота програми завершена")
```

Тобто в даному випадку спочатку перевіряється умова та виконуються інструкції while. Потім, коли умова стає рівною False, виконуються інструкції з блоку else. Зверніть увагу, що інструкції блоку else також мають відступи від початку конструкції циклу. У результаті ми отримаємо наступний консольний висновок:

```
number = 1
number = 2
number = 3
number = 4
number =5. Робота циклу завершена
Роботу програми завершено
```

Блок else може бути корисним, якщо умова спочатку дорівнює False, і ми можемо виконати деякі дії з цього приводу:

```
number = 10
while number < 5:
    print(f"number = {number}")
    number += 1
else:
    print(f"number = {number}. Робота циклу завершена")
print("Робота програми завершена")
```

В даному випадку умова `number < 5` спочатку дорівнює False, тому цикл не виконує жодної ітерації і відразу переходить до блоку else.

Цикл for

Інший тип циклів представляє конструкція for . Цей цикл пробігається по набору значень, поміщає кожне значення змінну, потім у циклі ми можемо з цієї змінної робити різні дії. Формальне визначення циклу for:

```
for змінна in набор_значень:
    інструкції
```

Після ключового слова for йде назва змінної, в яку будуть розміщуватись значення. Потім після оператора
in вказується набір значень та двокрапка.

А з наступного рядка розташовується блок інструкцій циклу, які повинні мати відступи від початку циклу.

При виконанні циклу Python послідовно отримує всі значення набору і передає їх змінну. Коли всі значення набору будуть перебрані, цикл завершує свою роботу.

Як набор значень, наприклад, можна розглядати рядок, який по суті представляє набір символів. Подивимося на прикладі:

```
message = "Hello"
for c in message:
    print(c)
```

У циклі визначається змінну `c` , після оператора `in` в якості набору, що перебирається, вказана змінна message, яка зберігає рядок "Hello". У результаті цикл for буде перебирати послідовно всі символи з рядка message і поміщати їх у змінну c. Блок самого циклу складається з однієї інструкції, яка виводить значення змінної з консоль. Консольний висновок програми:

```
H
e
l
l
o
```

Нерідко у зв'язці з циклом for застосовується вбудована функція range() , яка генерує числову послідовність:

```
for n in range(10):
    print(n, end=" ")
```

Якщо функцію range передається один параметр, він означає максимальне значення діапазону чисел. У цьому випадку генерується послідовність від 0 до 10 (не включно). У результаті ми отримаємо наступний консольний висновок:

```
0 1 2 3 4 5 6 7 8 9
```

Також у функцію range() можна передати мінімальне значення діапазону

```
for n in range(4, 10):
    print(n, end=" ")
```

Тут генерується послідовність від 4 до 10 (не включаючи). Консольний висновок:

```
4 5 6 7 8 9
```

Також у функцію range() можна передати третій параметр, який вказує на збільшення:

```
for n in range(0, 10, 2):
    print(n, end=" ")
```

Тут генерується послідовність від 0 до 10 (не включаючи) із збільшенням 2. Консольний висновок:

```
0 2 4 6 8
```

Цикл for також може мати додатковий блок else , який виконується після завершення циклу:

```
message = "Hello"
for c in message:
    print(c)
else:
    print(f"Останній символ: {c}. Цикл завершений");
print("Робота програми завершена")# інструкція не має відступу, тому не відноситься до else
```

В даному випадку ми отримаємо наступний консольний висновок:

```
H
e
l
l
o
Останній символ: o. Цикл завершено
Роботу програми завершено
```

Варто зазначити, що блок else має доступ до всіх змінних, визначених у циклі for.

Вкладені цикли

Одні цикли в собі можуть містити інші цикли. Розглянемо з прикладу виведення таблиці множення:

```
i = 1
j = 1
while i < 10:
    while j < 10:
        print(i * j, end="\t")
        j += 1
    print("\n")
    j = 1
    i += 1
```

Зовнішній цикл `while i < 10:` спрацьовує 9 разів поки змінна i не дорівнюватиме 10. Усередині цього циклу спрацьовує внутрішній цикл `while j < 10:`. Внутрішній цикл також спрацьовує 9 разів поки змінна j не дорівнюватиме 10. Причому всі 9 ітерацій внутрішнього циклу спрацьовують в рамках однієї ітерації зовнішнього циклу.

У кожній ітерації внутрішнього циклу на консоль виводиться добуток чисел i та j. Потім значення змінної j збільшується на одиницю. Коли внутрішній цикл закінчив роботу, значень змінної j скидається до 1, а значення змінної i збільшується на одиницю і відбувається перехід до наступної ітерації зовнішнього циклу. І все повторюється, поки змінна i не дорівнюватиме 10. Відповідно внутрішній цикл спрацює всього 81 раз для всіх ітерацій зовнішнього циклу. У результаті ми отримаємо наступний консольний висновок:

```
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
```

```
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

Подібним чином можна визначати вкладені цикли for:

```
for c1 in "ab":
    for c2 in "ba":
        print(f"{c1}{c2}")
```

У цьому випадку зовнішній цикл проходить рядком "ab" і кожен символ поміщає в змінну c1. Внутрішній цикл проходить рядком "ba", поміщає кожен символ рядка змінну c2 і виводить поєднання обох символів на консоль. Тобто в результаті ми отримаємо всі можливі поєднання символів a і b:

```
ab
aa
bb
ba
```

Вихід із циклу. break і continue

Для управління циклом ми можемо використовувати спеціальні оператори break та continue . Оператор break здійснює вихід із циклу. А оператор continue виконує перехід до наступної ітерації циклу.

Оператор break може використовуватися, якщо циклі утворюються умови, які несумісні з його подальшим виконанням. Розглянемо наступний приклад:

```
number = 0
while number < 5:
    number += 1
    if number == 3 :# якщо number = 3, виходимо з циклу
        break
    print(f"number = {number}")
```

Тут цикл while перевіряє умову `number < 5` . І поки number не дорівнює 5, передбачається, що значення number буде виводитися на консоль. Однак усередині циклу також перевіряється інша умова: `if number == 3` . Тобто, якщо значення number дорівнює 3, то за допомогою оператора break виходимо з циклу. І в результаті ми отримаємо наступний консольний висновок:

```
number = 1
```

```
number = 2
```

На відміну від оператора, `break` оператор continue виконує перехід до наступної ітерації циклу без його завершення. Наприклад, у попередньому прикладі замінимо break на continue:

```
number = 0
while number < 5:
    number += 1
    if number == 3 :# якщо number = 3, переходимо до нової ітерації
циклу
        continue
    print(f"number = {number}")
```

І в цьому випадку якщо значення змінної number дорівнює 3, наступні інструкції після оператора continue не виконуватимуться:

```
number = 1
number = 2
number = 4
number = 5
```

Додаткові матеріали

- Запитання для самоперевірки
- Вправи для самоперевірки

Функции
Функции представляют блок кода, который выполняет определенную задачу и который можно повторно использовать в других частях программы. В предыдущих
статьях уже использовались функции. В частности, функция print(), которая выводит некоторое значение на консоль. Python
имеет множество встроенных функций и позволяет определять свои функции. Формальное определение функции:

```
def имя_функции ([параметры]):
    инструкции
```

Определение функции начинается с выражения def, которое состоит из имени функции, набора скобок с параметрами и двоеточия.
Параметры в скобках необязательны. А со следующей строки идет блок инструкций, которые выполняет функция. Все инструкции функции имеют отступы от начала строки.

Например, определение простейшей функции:

```
def say_hello():
    print("Hello")
```

Функция называется `say_hello`. Она не имеет параметров и содержит одну единственную инструкцию,
которая выводит на консоль строку "Hello".

Обратите внимание, что инструкции функции должны иметь отступы от начала функции. Например:

```
def say_hello():
    print("Hello")


print("Bye")
```

Здесь инструкция `print("Bye")` не имеет отступов от начала функции say_hello и поэтому в эту функцию не входит.
Обычно между определением функции и остальными инструкциями, которые не входят в функцию, располагаются две пустых строки.

Для вызова функции указывается имя функции, после которого в скобках идет передача значений для всех ее параметров:

```
имя_функции ([параметры])
```

Например, определим и вызовем функцию:

```
def say_hello():   # определение функции say_hello
    print("Hello")


say_hello()        # вызов функции say_hello
say_hello()
say_hello()
```

Здесь три раза подряд вызывается функция say_hello. В итоге мы получим следующий консольный вывод:

```
Hello
Hello
Hello
```

Обратите внимание, что функция сначала определяется, а потом вызывается.

Если функция имеет одну инструкцию, то ее можно разместить на одной строке с остальным определением функции:

```
def say_hello(): print("Hello")

say_hello()
```

Подобным образом можно определять и вызывать и другие функции. Например, определим и выполним несколько функций:

```
def say_hello():
    print("Hello")


def say_goodbye():
    print("Good Bye")


say_hello()
say_goodbye()
```

Консольный вывод:

```
Hello
Good Bye
```

Локальные функции

Одни функции могут определяться внутри других функций - внутренние функции еще называют локальными. Локальные функции можно использовать только внутри той функции, в которой они определены. Например:

```
def print_messages():
   # определение локальных функций
    def say_hello(): print("Hello")
    def say_goodbye(): print("Good Bye")
   # вызов локальных функций
    say_hello()
    say_goodbye()

# Вызов функции print_messages
print_messages()
```

```
#say_hello()# вне функции print_messages функция say_hello не доступна
```

Здесь функции `say_hello()` и `say_goodbye()` определены внутри функции `print_messages()` и поэтому по отношению к ней
являются локальными. Соответственно они могут использоваться только внутри функции `print_messages()`

Организация программы и функция main

В программе может быть определено множество функций. И чтобы всех их упорядочить, одним из способов их организации является добавление специальной функции
(обычно называется `main`), в которой потом уже вызываются другие функции:

```
def main():
    say_hello()
    say_goodbye()

def say_hello():
    print("Hello")

def say_goodbye():
    print("Good Bye")

# Вызов функции main
main()
```

Параметры функции
Функция может принимать параметры. Через параметры в функцию можно передавать данные. Банальный пример - функция `print()`, которая с помощью параметра принимает значение, выводимое на консоль.

Теперь определим и используем свою функцию с параметрами:

```
def say_hello(name):
    print(f"Hello, {name}")

say_hello("Tom")
say_hello("Bob")
say_hello("Alice")
```

Функция say_hello имеет параметр name, и при вызове функции мы можем передать этому параметру какой-либо значение. Внутри функции мы можем

использовать
параметр как обычную переменную, например, вывести значение этого параметра на консоль функцией print. Так, в выражении:

```
say_hello("Tom")
```

Строка "Tom" будет передаваться параметру name. В итоге при выполнении программы мы получим следующий консольный вывод:

```
Hello, Tom
Hello, Bob
Hello, Alice
```

При вызове функции значения передаются параметрам по позиции. Например, определим и вызовем функцию с несколькими параметрами:

```
def print_person(name, age):
    print(f"Name: {name}")
    print(f"Age: {age}")


print_person("Tom", 37)
```

Здесь функция `print_person` принимает два параметра: name и age. При вызове функции:

```
print_person("Tom", 37)
```

Первое значение - "Tom" передается первому параметру, то есть параметру `name`. Второе значение - 37 передается второму параметру - age. И внутри функции значения параметров выводятся на консоль:

```
Name: Tom
Age: 37
```

Значения по умолчанию

Некоторые параметры функции мы можем сделать необязательными, указав для них значения по умолчанию при определении функции. Например:

```
def say_hello(name="Tom"):
    print(f"Hello, {name}")
```

```
say_hello()         # здесь параметр name будет иметь значение "Tom"
say_hello("Bob")    # здесь name = "Bob"
```

Здесь параметр name является необязательным. И если мы не передаем при вызове функции для него значение, то применяется значение по умолчанию, то есть
строка "Tom". Консольный вывод данной программы:

```
Hello, Tom
Hello, Bob
```

Если функция имеет несколько параметров, то необязательные параметры должны идти после обязательных. Например:

```
def print_person(name, age = 18):
    print(f"Name: {name}  Age: {age}")


print_person("Bob")
print_person("Tom", 37)
```

Здесь параметр age является необязательным и по умолчанию имеет значение 18. Перед ним расположен обязательный параметр name.
Поэтому при вызове функции мы можем не передавать значение параметру age, но параметру name передать значение необходимо.

При необходимости мы можем сделать все параметры необязательными:

```
def print_person(name = "Tom", age = 18):
    print(f"Name: {name}  Age: {age}")


print_person()              # Name: Tom  Age: 18
print_person("Bob")         # Name: Bob  Age: 18
print_person("Sam", 37)     # Name: Sam  Age: 37
```

Передача значений параметрам по имени. Именованные параметры

В примерах выше при вызове функции значения передаются параметрами функции по позиции. Но также можно передавать значения параметрам по имени. Для этого при вызове
функции указывается имя параметра и ему присваивается значение:

```
def print_person(name, age):
    print(f"Name: {name}  Age: {age}")


print_person(age = 22, name = "Tom")
```

В данном случае значения параметрам age и name передаются по имени. И несмотря на то,
что параметр name идет первым в определении функции, мы можем при вызове функции написать `print_person(age = 22, name = "Tom")` и таким образом передать число 22 параметру age, а строку "Tom" параметру name.

Символ * позволяет установить, какие параметры будут именнованными - то есть такие параметры, которым можно передать значения
только по имени. Все параметры, которые располагаются справа от символа *, получают значения только по имени:

```
def print_person(name, *,  age, company):
    print(f"Name: {name}  Age: {age}  Company: {company}")


print_person("Bob", age = 41, company ="Microsoft")   # Name: Bob
Age: 41  company: Microsoft
```

В данном случае параметры age и company являются именнованными.

Можно сделать все параметры именнованными, поставив перед списком параметров символ *:

```
def print_person(*,  name, age, company):
    print(f"Name: {name}  Age: {age}  Company: {company}")
```

Если наоборот надо определить параметры, которым можно передавать значения только по позиции, то есть позиционные параметры,
то можно использовать символ /: все параметры, которые идут до символа / , являются позиционными и могут получать значения только по позиции

```
def print_person(name, /, age, company="Microsoft"):
    print(f"Name: {name}  Age: {age}  Company: {company}")


print_person("Tom", company="JetBrains", age = 24)    # Name: Tom
Age: 24  company: JetBrains
print_person("Bob", 41)                # Name: Bob  Age: 41  company:
Microsoft
```

В данном случае параметр name является позиционным.

Для одной функции можно определять одновременно позиционные и именованные параметры.

```
def print_person(name, /,  age = 18, *, company):
    print(f"Name: {name}  Age: {age}  Company: {company}")


print_person("Sam", company ="Google")              # Name: Sam  Age:
18  company: Google
print_person("Tom", 37, company ="JetBrains")       # Name: Tom  Age:
37  company: JetBrains
print_person("Bob", company ="Microsoft", age = 42) # Name: Bob  Age:
42  company: Microsoft
```

В данном случае параметр `name` располагается слева от символа /, поэтому является позиционным и обязательным - ему можно
передать значение только по позиции.

Параметр `company` является именованным, так как располагается справа от символа *. Параметр `age` может получать значение по имени и по позиции.

Неопределенное количество параметров

С помощью символа звездочки можно определить параметр, через который можно передавать неопределенное количество значений. Это может быть полезно,
когда мы хотим, чтобы функция получала несколько значений, но мы точно не знаем, сколько именно. Например, определим функцию подсчета суммы чисел:

```
def sum(*numbers):
    result = 0
    for n in numbers:
        result += n
    print(f"sum = {result}")


sum(1, 2, 3, 4, 5)      # sum = 15
sum(3, 4, 5, 6)         # sum = 18
```

В данном случае функция sum принимает один параметр - `*numbers`, но звездочка перед названием параметра указывает, что фактически на место этого
параметра мы можем передать неопределенное количество значений или

набор значений. В самой функции с помощью цикла for можно пройтись по этому набору,
получить каждое значение из этого набора в переменную `n` и произвести с ним какие-нибудь действия. Например, в данном случае вычисляется сумма переданных чисел.

Дополнительные материалы

- Вопросы для самопроверки

Оператор return та повернення результату з функції

Повернення результату

Функція може повертати результат. Для цього в функції використовується оператор return , після якого вказується значення, що повертається:

```
def ім'я_функції ([параметри]):
    інструкції
    return повертається значення
```

Визначимо найпростішу функцію, яка повертає значення:

```
def get_message():
    return "Hello METANIT.COM"
```

Тут після оператора return йде рядок `"Hello METANIT.COM"` - це значення і повертатиме функція
`get_message()` .

Потім цей результат функції можна присвоїти змінній або використовувати як звичайне значення:

```
def get_message():
    return "Hello METANIT.COM"

message = get_message()# отримуємо результат функції get_message до змінної message
print(message)# Hello METANIT.COM
# можна безпосередньо передати результат функції get_message
print(get_message())# Hello METANIT.COM
```

Після оператора return може йти і складний вираз, результат якого буде повертатися з функції. Наприклад, визначимо функцію, яка збільшує число вдвічі:

```
def double(number):
    return 2 * number
```

Тут функція double повертатиме результат виразу `2 * number`:

```
def double(number):
    return 2 * number
result1 = double(4)# result1 = 8
result2 = double(5)# result2 = 10
print(f"result1 = {result1}")# result1 = 8
print(f"result2 = {result2}")# result2 = 10
```

Або інший приклад – отримання суми чисел:

```
def sum(a, b):
    return a + b

result = sum(4, 6)# result = 0
print(f"sum(4, 6) = {result}")# sum(4, 6) = 10
print(f"sum(3, 5) = {sum(3, 5)}")# sum(3, 5) = 8
```

Вихід із функції

Оператор return як повертає значення, а й виробляє вихід із функції. Тому він має визначатися після решти інструкцій. Наприклад:

```
def get_message():
    return "Hello METANIT.COM"
    print("End of the function")
print(get_message())
```

З погляду синтаксису дана функція коректна, проте її інструкція `print("End of the function")` не має сенсу - вона ніколи не виконається, тому що до її виконання оператор return поверне значення та здійснить вихід із функції.

Однак ми можемо використовувати оператор return і в таких функціях, які не повертають жодного значення. У цьому випадку після оператора return не ставиться ніякого значення, що повертається. Типова ситуація - залежно від випереджених умов зробити вихід із функції:

```
def print_person(name, age):
    if age > 120 or age < 1:
        print("Invalid age")
        return
```

```
    print(f"Name: {name} Age: {age}")

print_person("Tom", 22)
print_person("Bob", -102)
```

Тут функція `print_person` як параметри приймає ім'я та вік користувача. Однак у функції спочатку ми перевіряємо, чи вік відповідає деякому діапазону (менше 120 і більше 0). Якщо вік знаходиться поза цим діапазоном, то виводимо повідомлення про неприпустимий вік і за допомогою оператора return виходимо з функції. Після цього функція закінчує свою роботу.

Однак якщо вік коректний, виводимо інформацію про користувача на консоль. Консольний висновок:

```
Name: Tom Age: 22
Invalid age
```

Функція як тип, параметр та результат іншої функції

Функція як тип

У Python функція фактично представляє окремий тип. Так ми можемо присвоїти змінній якусь функцію і потім, використовуючи змінну, викликати цю функцію. Наприклад:

```
def say_hello(): print("Hello")
def say_goodbye(): print("Good Bye")
message=say_hello
message()# Hello
message=say_goodbye
message()# Good Bye
```

У разі змінної `message` присвоюється одне з функцій. Спочатку їй передається функція `say_hello()`:

```
message=say_hello
```

Після цього змінна message вказуватиме на цю функцію, тобто фактично представляти функцію say_hello. А це означає, що ми можемо викликати змінну message як звичайну функцію:

```
message()# Hello
```

Фактично це призведе до виконання функції say_hello, і на консоль буде виведено рядок Hello. Потім ми можемо передати змінній message іншу функцію

і викликати її.

Подібним чином можна через змінну викликати функцію з параметрами та повертати її результат:

```
def sum(a, b): return a + b
def multiply(a, b): return a * b
operation = sum
result = operation(5, 6)
print(result)# 11
operation = multiply
print(operation(5, 6))# 30
```

Функція як параметр функції

Оскільки функція Python може представляти таке ж значення як рядок або число, відповідно ми можемо передати її як параметр в іншу функцію. Наприклад, визначимо функцію, яка виводить на консоль результат деякої операції:

```
def do_operation(a, b, operation):
    result = operation(a, b)
    print(f"result = {result}")
def sum(a, b): return a + b
def multiply(a, b): return a * b
do_operation(5, 4, sum)# result = 9
do_operation(5, 4, multiply)# result = 20
```

У даному випадку функція do_operation має три параметри, причому третій параметр, як передбачається, представлятиме функцію, яка приймає два параметри та повертає деякий результат. Іншими словами третій параметр - operation представляє деяку операцію, але на момент визначення функції do_operation ми точно не знаємо, що це буде за операція. Ми тільки знаємо, що вона приймає два параметри і повертає якийсь результат, який потім виводиться на консоль.

При виклику функції do_operation ми зможемо передати як третій параметр іншу функцію, наприклад, функцію sum:

```
do_operation(5, 4, sum)
```

Тобто в даному випадку параметр operation фактично представлятиме функцію sum і повертатиме суму двох чисел.

Потім аналогічним образам виклик функції do_operation можна передати третьому параметру іншу функцію - multiply, яка виконає множення чисел:

```
do_operation(5, 4, multiply)# result = 20
```

Таким чином, гнучкіші за функціональністю функції, які через параметри приймають інші функції.

Функція як результат функції

Також одна функція Python може повертати іншу функцію. Наприклад, визначимо функцію, яка в залежності від значення параметра повертає ту чи іншу операцію:

```
def sum(a, b): return a + b
def subtract(a, b): return a - b
def multiply(a, b): return a * b

def select_operation(choice):
    if choice == 1:
        return sum
    elif choice == 2:
        return subtract
    else:
        return multiply

operation = select_operation(1)# operation = sum
print(operation(10, 6))# 16
operation = select_operation(2)# operation = subtract
print(operation(10, 6))# 4
operation = select_operation(3)# operation = multiply
print(operation(10, 6))# 60
```

В даному випадку функція select_operation в залежності від значення параметра choice повертає одну з трьох функцій – sum, subtract та multiply. Потім ми можемо отримати результат функції select_operation у змінну operation:

```
operation = select_operation(1)
```

Так, в даному випадку в функцію select_operation передається число 1, відповідно вона повертатиме функцію sum. Тому змінна operation фактично вказуватиме на функцію sum, яка виконує додавання двох чисел:

```
print(operation(10, 6))# 16 - фактично одно sum(10, 6)
```

Аналогічним чином можна отримати та виконати інші функції.

Лямбда-вирази
Лямбда-вирази в мові Python є невеликими анонімними функціями, які визначаються за допомогою оператора lambda . Формальне визначення лямбда-виразу:

```
lambda [параметри] : інструкція
```

Визначимо найпростіший лямбда-вираз:

```
message = lambda: print("hello")
message()# hello
```

Тут лямбда-вираз присвоюється змінній message. Цей лямбда-вираз не має параметрів, нічого не повертає і просто виводить рядок "hello" на консоль. Та через мінливу message ми можемо викликати це лямбда-вираз як звичайну функцію. Фактично воно аналогічне наступній функції:

```
def message():
    print("hello")
```

Якщо лямбда-вираз має параметри, вони визначаються після ключового слова lambda . Якщо лямбда-вираз повертає якийсь результат, він вказується після двокрапки. Наприклад, визначимо лямбда-вираз, який повертає квадрат числа:

```
square = lambda n: n * n
print(square(4))# 16
print(square(5))# 25
```

В даному випадку лямбда-вираз приймає один параметр – n. Праворуч від двокрапки йде значення, що повертається - `n* n` . Це лямбда-вираз аналогічно до наступної функції:

```
def square2(n): return n * n
```

Аналогічно можна створювати лямбда-вирази, які приймають кілька параметрів:

```
sum = lambda a, b: a + b
print(sum(4, 5))# 9
print(sum(5, 6))# 11
```

Хоча лямбда-вирази дозволяють трохи скоротити визначення функцій, проте вони обмежені тим, що вони можуть виконувати лише один вираз. Однак вони можуть бути досить зручні в тих випадках, коли необхідно використовувати функцію передачі як параметра або повернення в іншій функції. Наприклад, передача лямбда-виразу як параметр:

```
def do_operation(a, b, operation):
    result = operation(a, b)
    print(f"result = {result}")
do_operation(5, 4, lambda a, b: a + b)# result = 9
do_operation(5, 4, lambda a, b: a * b)# result = 20
```

В даному випадку нам немає необхідності визначати функції, щоб передати їх як параметр, як у попередній статті.

Те саме стосується й повернення лямбда-виразів із функцій:

```
def select_operation(choice):
    if choice == 1:
        return lambda a, b: a + b
    elif choice == 2:
        return lambda a, b: a - b
    else:
        return lambda a, b: a * b

operation = select_operation(1)# operation = sum
print(operation(10, 6))# 16
operation = select_operation(2)# operation = subtract
print(operation(10, 6))# 4
operation = select_operation(3)# operation = multiply
print(operation(10, 6))# 60
```

Перетворення типів
У операціях з даними можуть застосовуватися значення різних типів. Наприклад, складаються число типу `int` та число типу `float`:

```
a = 2# число int
b = 2.5# число float
c = a + b
print(c)# 4.5
```

В даному випадку жодної помилки не буде. Однак Python не завжди може автоматично виконувати операції, в яких беруть участь дані різних типів. Розглянемо, які у разі діють правила.

Неявні перетворення

Обидва числа в арифметичних операціях повинні представляти той самий тип. Якщо ж два операнди операції представляють різні типи даних, то Python намагається автоматично виконати перетворення до одного з типів відповідно до наступних правил:

- Якщо з операндов операції представляє комплексне число (тип complex), інший операнд також перетворюється на тип complex.
- Інакше, якщо один з операндів представляє тип float , то другий операнд також перетворюється на тип float. Власне так і сталося в прикладі вище, де значення змінної `a` було перетворено на тип float
- Інакше, обидва операнди повинні представляти тип int , і в цьому випадку переобазування не потрібно

Явні перетворення

Але в деяких випадках виникає потреба вручну виконати перетворення типів. Наприклад, нехай у нас буде наступний код:

```
a = "2"
b = 3
c = a + b
```

Ми очікуємо, що "2" + 3 дорівнюватиме 5. Однак цей код згенерує виняток, так як перше число насправді представляє рядок. І ми побачимо при виконанні коду щось на кшталт:

```
Traceback (most recent call last):
  File "/Users/Nikita/PycharmProjects/HelloApp/main.py", line 3, in
    c = a + b
TypeError: може лише написати str (не "int") to str
```

Для перетворення типів Python надає ряд вбудованих функцій:

- int() : перетворює значення на ціле число
- float() : перетворює значення на число з плаваючою точкою
- str() : перетворює значення на рядок

int

Так, у попередньому прикладі перетворюємо рядок на число за допомогою функції int():

```
a = "2"
b = 3
c = int(a) + b
print(c)# 5
```

Приклади перетворень за допомогою `int()`:

```
a = int(15)# a = 15
b = int (3.7)# b = 3
c = int("4")# c = 4
e = int (False)# e = 0
f = int (True)# f = 1
```

Однак якщо значення не може бути перетворено, функція int видасть помилку `ValueError: invalid literal for int() with base 10`:

```
b = int("a1c")# Помилка
c = int("4.7")# Помилка
```

float

Аналогічним чином діє функція float() , яка перетворює на число з плаваючою точкою.

```
a = "2.7"
b = 3
c = float(a) + b
print(c)# 5.7
```

Приклади перетворень за допомогою `float()`:

```
a = float(15)# a = 15.0
b = float(3.7)# b = 3.7
c = float("4.7")# c = 4.7
d = float("5")# d = 5.0
e = float (False)# e = 0.0
f = float (True)# f = 1.0
```

Але знову ж таки не всі значення можуть автоматично перетворені на float. Так, у наступному випадку Python згенерує помилку:

```
d = float("abc")# Помилка
```

str

Функція str() перетворює значення на рядок:

```
a = str (False)# a = "False"
b = str (True)# b = "True"
c = str (5)# c = "5"
d = str (5.7)# d = "5.7"
```

Функція `str()` може бути актуальною, наприклад, при додаванні до рядка значення іншого типу. Наприклад, у наступному випадку ми отримаємо помилку:

```
age = 22
message = "Age:" + age# Помилка
print(message)
```

Якщо число складається з число, це стандартна операція складання чисел. Якщо рядок складається з рядком, це операція об'єднання рядків. Але яким чином виконати операцію додавання до рядка і числа, Python не знає. І якщо ми в даному випадку хочемо виконати операцію об'єднання рядків, то число можна привести до рядка за допомогою функції
`str()`:

```
age = 22
message = "Age:" + str(age)# Age: 22
print(message)
```

Область видимості змінних
Область видимості чи scope визначає контекст змінної, у якого її можна використовувати. У Python є два типи контексту: глобальний та локальний.

Глобальний контекст

Глобальний контекст має на увазі, що змінна є глобальною, вона визначена поза будь-якою функцією і доступна будь-якій функції в програмі. Наприклад:

```
name = "Tom"

def say_hi():
    print("Hello", name)

def say_bye():
    print("Good bye", name)
```

```
say_hi()
say_bye()
```

Тут змінна назва є глобальною і має глобальну область видимості. І обидві певні функції можуть вільно її використовувати.

Локальний контекст

На відміну від глобальних змінних, локальна змінна визначається всередині функції і доступна тільки з цієї функції, тобто має локальну область видимості:

```
def say_hi():
    name = "Sam"
    surname = "Johnson"
    print("Hello", name, surname)

def say_bye():
    name = "Tom"
    print("Good bye", name)
say_hi()
say_bye()
```

У разі кожної з двох функцій визначається локальна змінна name. І хоча ці змінні називаються однаково, проте це дві різних змінних, кожна з яких доступна тільки в рамках своєї функції. Також у функції `say_hi()` визначена змінна surname, яка також є локальною, тому функції `say_bye()` ми її використовувати не зможемо.

Приховування змінних

Є ще один варіант визначення змінної, коли локальна змінна приховують глобальну з тим самим ім'ям:

```
name = "Tom"

def say_hi():
    name = "Bob"# приховуємо значення глобальної змінної
    print("Hello", name)

def say_bye():
    print("Good bye", name)

say_hi()# Hello Bob
say_bye()# Good bye Tom
```

Тут визначено глобальну змінну назву. Однак функції say_hi визначена локальна змінна з тим же ім'ям name. І якщо функція say_bye використовує глобальну змінну, функція say_hi використовує локальну змінну, яка приховує глобальну змінну.

Якщо ми хочемо змінити в локальної функції глобальну змінну, а чи не визначити локальну, необхідно використовувати ключове слово global :

```
name = "Tom"

def say_hi():
    global name
    name = "Bob"# змінюємо значення глобальної змінної
    print("Hello", name)

def say_bye():
    print("Good bye", name)

say_hi()# Hello Bob
say_bye()# Good bye Bob
```

nolocal

Вираз nonlocal прикріплює ідентифікатор до змінної з найближчого навколишнього контексту (крім глобального контексту). Зазвичай nonlocal застосовується у вкладених функціях, коли треба прикріпити ідентифікатор за змінною або параметром зовнішньої зовнішньої функції. Розглянемо ситуацію, де цей вираз може стати в нагоді:

```
def outer():# зовнішня функція
    n = 5
    def inner():# вкладена функція
        print(n)
    inner()# 5
    print(n)

outer()# 5
```

Тут вкладена локальна функція inner() виводить на консоль значення змінної n яка визначена у зовнішній функції outer(). Потім функції outer() викликається внутрішня функція inner().

При виклику функції outer() тут ми очікуємо побачимо на консолі двічі число 5. Однак у разі вкладена функція inner() просто отримує значення. Тепер візьмемо

іншу ситуацію, коли вкладена функція надає значення змінної:

```
def outer():# зовнішня функція
    n = 5
    def inner():# вкладена функція
        n = 25
        print(n)
    inner()# 25
    print(n)

outer()# 5
# 25 - inner
# 5 - outer
```

При наданні значення у вкладеній функції: `n = 25` буде створюватися нова змінна n, яка приховає змінну n з зовнішньої зовнішньої функції outer. У результаті отримаємо при виведенні два різних числа. Щоб у вкладеній функції вказати, що ідентифікатор у вкладеній функції представлятиме змінну з навколишньої функції, застосовується вираз nonlocal :

```
def outer():# зовнішня функція
    n = 5
    def inner():# вкладена функція
        nonlocal n# вказуємо, що n - це змінна з навколишньої функції
        n = 25
        print(n)
    inner()# 25
    print(n)

outer()# 25
```

Замикання
Замикання (closure) представляє функцію, яка запам'ятовує своє лексичне оточення навіть у тому випадку, коли вона виконується поза своєю областю видимості.

Технічно замикання включає три компоненти:

- зовнішня функція, яка визначає деяку область видимості та в якій визначені деякі змінні та параметри - лексичне оточення
- змінні та параметри (лексичне оточення), які визначені у зовнішній функції
- вкладена функція, яка використовує змінні та параметри зовнішньої функції

Для визначення замикань у Python застосовуються локальні функції:

```
def outer():# зовнішня функція
    n = 5# лексичне оточення - локальна змінна
    def inner():# локальна функція
        nonlocal n
        n += 1# операції з лексичним оточенням
        print(n)
    return inner

fn = outer()# fn = inner, тому що функція outer повертає функцію
inner
# Викликаємо внутрішню функцію inner
fn()# 6
fn()# 7
fn()# 8
```

Тут функція `outer` визначає локальну змінну n - і є лексичне оточення для внутрішньої функції:

Усередині функції outer визначено внутрішню функцію - локальну функцію inner, яка звертається до свого лексичного оточення - змінної n - збільшує її значення на одиницю і виводить на консоль:

```
def inner():# локальна функція
    nonlocal n
    n += 1# операції з лексичним оточенням
    print(n)
```

Ця локальна функція повертається функцією outer:

```
return inner
```

У програмі викликаємо функцію outer і отримуємо змінну `fn` локальну функцію inner:

```
fn = outer()
```

Змінна `fn` і є замикання, тобто поєднує дві речі: функцію і оточення, в якому функція була створена. І незважаючи на те, що ми отримали локальну функцію і можемо її викликати поза її навколишньою функцією, в якій вона визначена, проте вона запам'ятала своє лексичне оточення і може до нього звертатися та змінювати, що ми побачимо з консольного висновку:

```
fn()# 6
fn()# 7
```

```
fn()# 8
```

Застосування параметрів

Крім зовнішніх змінних до лексичного оточення також належать параметри довкілля. Розглянемо використання параметрів:

```
def multiply(n):
    def inner(m): return n * m
    return inner

fn = multiply(5)
print(fn(5))# 25
print(fn(6))# 30
print(fn(7))# 35
```

Тут зовнішня функція - multiply повертає функцію, яка приймає число та повертає число.

Виклик функції `multiply()` повертає локальну функцію inner:

```
def inner(m): return n * m
```

Ця функція запам'ятовує оточення, де вона була створена, зокрема, значення параметра n. Крім того, сама приймає параметр і повертає добуток параметрів n та m.

У результаті виклику функції multiply визначається змінна fn, яка отримує локальну функцію inner та її лексичне оточення - значення параметра n:

```
fn = multiply(5)
```

У разі параметр n дорівнює 5.

При виклику локальної функції, наприклад, у разі:

```
print(fn(6))# 30
```

Число 6 передається для параметра m локальної функції, яка повертає добуток n та m, тобто 5 * 6 = 30.

Також можна було б скоротити цей код за допомогою лямбд:

```
def multiply(n): return lambda m: n * m

fn = multiply(5)
print(fn(5))# 25
print(fn(6))# 30
print(fn(7))# 35
```

Декоратори

Декоратори в Python представляють функцію, яка як параметр отримує функцію і як результат також повертає функцію. Декоратори дозволяють модифікувати функцію, значення її параметрів і її результат без зміни вихідного коду цієї функції.

Розглянемо найпростіший приклад:

```
визначення функції декоратора
def select(input_func):
    def output_func():# визначаємо функцію, яка виконуватиметься
замість оригінальної
        print("*****************")# перед виведенням оригінальної
функції виводимо всяку зірочки
        input_func()# виклик оригінальної функції
        print("*****************")# після виведення оригінальної
функції виводимо всяку зірочки
    return output_func# повертаємо нову функцію
визначення оригінальної функції
@select# застосування декоратора select
def hello():
    print("Hello METANIT.COM")
виклик оригінальної функції
hello()
```

Спочатку визначається власне функція декоратора, яка у разі називається `select()`. Як параметр декоратор отримує функцію (в даному випадку параметр
`input_func`), до якої цей декоратор застосовуватиметься:

```
def select(input_func):
    def output_func():# визначаємо функцію, яка виконуватиметься
замість оригінальної
        print("*****************")# перед виведенням оригінальної
функції виводимо всяку зірочки
        input_func()# виклик оригінальної функції
        print("*****************")# після виведення оригінальної
функції виводимо всяку зірочки
    return output_func# повертаємо нову функцію
```

Результатом декоратора у разі є локальна функція `output_func`, у якій викликається вхідна функція input_func. Для простоти тут перед і після виклик input_func для краси просто виводимо набір символів "*".

Далі визначається стандартна функція, до якої застосовується декоратор - в даному випадку це функція `hello`, яка просто виводить на консоль деякий рядок:

```
@select# застосування декоратора select
def hello():
    print("Hello METANIT.COM")
```

Для застосування декоратора перед визначенням функції вказується символ `@`, після якого йде ім'я декоратора. Тобто в цьому випадку до функції hello() застосовується декоратор select().

Далі викликаємо звичайну функцію:

```
hello()
```

Оскільки до цієї функції застосовується декоратор select, то в результаті функція hello передається в декоратор `select()` як параметр `input_func`. І оскільки декоратор повертає нову функцію – output_func, то фактично в даному випадку виконуватиметься саме ця функція `output_func()`

У результаті ми отримаємо наступний консольний висновок:

```
**********
Hello METANIT.COM
**********
```

Отримання параметрів функції у декораторі

Декоратор може перехоплювати аргументи, що передаються в функцію:

```
визначення функції декоратора
def check(input_func):
    def output_func(*args):# через *args отримуємо значення
параметрів оригінальної функції
        input_func(*args)# виклик оригінальної функції
    return output_func# повертаємо нову функцію
визначення оригінальної функції
@check
def print_person(name, age):
    print(f"Name: {name} Age: {age}")
```

```
виклик оригінальної функції
print_person("Tom", 38)
```

Тут функція `print_person()` приймає два параметри: name (ім'я) та age (вік). До цієї функції застосовується декоратор `check()`

У декораторі `check` повертається локальна функція `output_func()`, яка набирає певний набір значень як параметра `*args` - це значення, які передаються в оригінальну функцію, до якої застосовується декоратор. Тобто в даному випадку `*args` міститиме значення параметрів name та age.

```
def check(input_func):
    def output_func(*args):# через *args отримуємо значення
параметрів функції input_func
```

Тут просто передаємо ці значення в оригінальну функцію:

```
input_func(*args)
```

У результаті в цьому отримаємо наступний консольний висновок

```
Name: Tom Age: 38
```

Але що якщо в функцію print_person буде передано якесь неприпустиме значення, наприклад, негативний вік? Однією з переваг декораторів є те, що ми можемо перевірити і при необхідності модифікувати значення параметрів. Наприклад:

```
визначення функції декоратора
def check(input_func):
    def output_func(*args):
        name = args[0]
        age = args[1]# отримуємо значення другого параметра
        if age < 0: age = 1# якщо вік негативний, змінюємо його
значення на 1
        input_func(name, age)# передаємо функції значення параметрів
    return output_func
визначення оригінальної функції
@check
def print_person(name, age):
    print(f"Name: {name} Age: {age}")
виклик оригінальної функції
print_person("Tom", 38)
print_person("Bob", -5)
```

args фактично представляє набір значень, і, використовуючи індекси, ми можемо отримати значення параметрів за позицією і щось з ними зробити. Так, тут, якщо значення віку менше 0, то встановлюємо 1. Потім передаємо ці значення виклик функції. У результаті тут отримаємо такий висновок:

```
Name: Tom Age: 38
Name: Bob Age: 1
```

Отримання результату функції

Подібним чином можна отримати результат функції та за необхідності змінити його:

```
визначення функції декоратора
def check(input_func):
    def output_func(*args):
        result = input_func(*args)# передаємо функції значення
параметрів
        if result < 0: result = 0# якщо результат функції менший за
нуль, то повертаємо 0
        return result
    return output_func
визначення оригінальної функції
@check
def sum(a, b):
    return a + b
виклик оригінальної функції
result1 = sum(10, 20)
print(result1)# 30
result2 = sum(10, -20)
print(result2)# 0
```

Тут визначено функцію `sum()`, яка повертає суму чисел. У декораторі check перевіряємо результат функції і для простоти, якщо він менший за нуль, то повертаємо 0.

Консольний висновок програми:

```
30
0
```

Розділ 3. Об'єктно-орієнтоване програмування

Класи та об'єкти

Python має безліч вбудованих типів, наприклад, int, str і так далі, які ми можемо використовувати у програмі. Але також Python дозволяє визначати власні типи за допомогою класів . Клас є деякою сутністю. Конкретним здійсненням класу є об'єкт.

Можна ще провести таку аналогію. У нас у всіх є деяке уявлення про людину, яка має ім'я, вік, якісь інші характеристики. Людина може виконувати деякі дії – ходити, бігати, думати тощо. Тобто це уявлення, яке включає набір характеристик та дій, можна назвати класом. Конкретне втілення цього шаблону може відрізнятися, наприклад, одні мають одне ім'я, інші - інше ім'я. І реально існуюча людина представлятиме об'єкт цього класу.

У мові Python клас визначається за допомогою ключового слова .

```
class назва_класу:
    атрибути_класу
    методи_класу
```

Усередині класу визначаються його атрибути, які зберігають різні характеристики класу, та методи – функції класу.

Створимо найпростіший клас:

```
class Person:
    pass
```

У разі визначено клас Person, який умовно представляє людини. В даному випадку в класі не визначається жодних методів чи атрибутів. Однак оскільки в ньому має бути щось визначено, то як замінник функціоналу класу застосовується оператор pass . Цей оператор застосовується, коли синтаксично необхідно визначити певний код, проте виходячи із завдання код нам не потрібен, і замість конкретного коду вставляємо оператор pass.

Після створення класу, можна визначити об'єкти цього класу. Наприклад:

```
class Person:
    pass
tom = Person()# визначення об'єкта tom
bob = Person()# визначення об'єкта bob
```

Після визначення класу Person створюються два об'єкти класу Person – tom і bob. Для створення об'єкта застосовується спеціальна функція - конструктор , яка називається на ім'я класу і яка повертає об'єкт класу. Тобто у цьому випадку виклик `Person()` представляє виклик конструктора. Кожен клас за замовчуванням має конструктор без параметрів:

```
tom = Person()# Person() - виклик конструктора, який повертає об'єкт класу Person
```

Конструктори

Отже, створення об'єкта класу використовується конструктор. Так, вище коли ми створювали об'єкти класу Person, ми використовували за замовчуванням конструктор, який не приймає параметрів і який неявно мають всі класи. Однак ми можемо явно визначити в класах конструктор за допомогою спеціального методу, який називається __init__() (по два прочерки з кожного боку). Наприклад, змінимо клас Person, додавши до нього конструктор:

```
class Person:
   # конструктор
    def __init__(self):
        print("Створення об'єкта Person")
tom = Person()# Створення об'єкта Person
```

Отже, тут у коді класу Person визначений конструктор – функція __init__ . Конструктор повинен приймати щонайменше один параметр посилання на поточний об'єкт- self . Зазвичай конструктори застосовуються визначення дій, які мають здійснюватися під час створення об'єкта.

Тепер під час створення об'єкта:

```
tom = Person()
```

Виконується виклик конструктора `__init__()` з класу Person, який виведе на консоль рядок "Створення об'єкта Person".

Конструктор фактично представляє звичайну функцію, тільки для виклику конструктора використовується не `__init__` , а назва класу. Крім того, при виклику конструктора параметру self явно не передається ніякого значення. При виконанні програми Python динамічно визначатиме self .

Атрибути об'єкту

Атрибути зберігають стан об'єкта. Для визначення та встановлення атрибутів усередині класу можна застосовувати слово self . Наприклад, визначимо наступний клас Person:

```
class Person:
    def __init__(self, name, age):
```

```
        self.name = name# ім'я людини
        self.age = age# вік людини

tom = Person("Tom", 22)
звернення до атрибутів
# Отримання значень
print(tom.name)# Tom
print(tom.age)# 22
# Зміна значення
tom.age = 37
print(tom.age)# 37
```

Тепер конструктор класу Person приймає ще два параметри – name та age. Через ці параметри в конструктор будуть передаватися ім'я та вік людини, що створюється.

Усередині конструктора встановлюються два атрибути - name і age (умовно ім'я та вік людини):

```
def __init__(self, name, age):
    self.name = name
    self.age = age
```

Атрибуту self.name надається значення змінної name. Атрибут age набуває значення параметра age. Назва атрибутів не обов'язково має відповідати назвам параметрів.

Якщо ми визначили в класі конструктор __init__ із параметрами (крім self), то при виклику конструктора цим параметрам треба передати значення:

```
tom = Person("Tom", 22)
```

Тобто в даному випадку параметр name передається рядок "Tom", а параметр age - число 22.

Далі на ім'я об'єкта ми можемо звертатися до атрибутів об'єкта - отримувати та змінювати їх значення:

```
print(tom.name)# Отримання значення атрибута name
tom.age = 37# зміна значення атрибуту age
```

Подібним чином ми можемо створювати різні об'єкти класу Person із різним значенням для атрибутів:

```
class Person:
    def __init__(self, name, age):
        self.name = name# ім'я людини
        self.age = age# вік людини

tom = Person("Tom", 22)
bob = Person("Bob", 43)
print(tom.name)# Tom
print(bob.name)# Bob
```

Тут створюються два об'єкти класу Person: tom та bob. Вони відповідають визначенню класу Person, мають однаковий набір атрибутів, однак їхній стан відрізнятиметься. І в кожному випадку Python динамічно визначатиме об'єкт self. Так, у наступному випадку

```
tom = Person("Tom", 22)
```

Це буде об'єкт tom

А під час виклику

```
bob = Person("Bob", 43)
```

Це буде об'єкт bob

В принципі нам необов'язково визначати атрибути всередині класу - Python дозволяє зробити це динамічно поза кодом:

```
class Person:
    def __init__(self, name, age):
        self.name = name# ім'я людини
        self.age = age# вік людини

tom = Person("Tom", 22)
tom.company = "Microsoft"
print(tom.company)# Microsoft
```

Тут динамічно встановлюється атрибут company, який зберігає місце роботи людини. І після встановлення ми також можемо набути його значення. У той же час подібне визначення загрожує помилками. Наприклад, якщо ми спробуємо звернутися до атрибуту до його визначення, програма згенерує помилку:

```
tom = Person("Tom", 22)
print(tom.company)#! Помилка - AttributeError: Person object не має
```

```
attribute company
```

Методи класів

Методи класу фактично представляють функції, які визначені всередині класу та визначають його поведінку. Наприклад, визначимо клас Person з одним методом:

```
class Person:# визначення класу Person
     def say_hello(self):
        print("Hello")
tom = Person()
tom.say_hello()# Hello
```

Тут визначено метод `say_hello()`, який умовно виконує вітання – виводить рядок на консоль. При визначенні методів будь-якого класу, як і конструктора, перший параметр методу є посиланням на поточний об'єкт, який згідно з умовностями називається self . Через це посилання всередині класу ми можемо звернутися до функціональності об'єкта. Але при самому виклик методу цей параметр не враховується.

Використовуючи ім'я об'єкта, ми можемо звернутися до його способів. Для звернення до методів застосовується нотація точки – після імені об'єкта ставиться точка і після неї йде виклик методу:

```
об'єкт.метод([параметри методу])
```

Наприклад, звернення до методу `say_hello()` виведення вітання на консоль:

```
tom.say_hello()# Hello
```

У результаті ця програма виведе на консоль рядок "Hello".

Якщо метод повинен приймати інші параметри, то вони визначаються після параметра `self` і при виклику подібного методу для них необхідно передати значення:

```
class Person:# визначення класу Person
    def say(self, message):# метод
        print(message)
tom = Person()
tom.say("Hello METANIT.COM")# Hello METANIT.COM
```

Тут визначено метод `say()`. Він приймає два параметри: self і message. І другого параметра - message при виклику методу необхідно передати значення.

Для звернення до атрибутів та методів об'єкта всередині класу у його методах також застосовується слово self:

```
self.атрибут# звернення до атрибуту
self.метод# звернення до методу
```

Наприклад, наступний клас Person:

```
class Person:
    def __init__(self, name, age):
        self.name = name# ім'я людини
        self.age = age# вік людини

    def display_info(self):
        print(f"Name: {self.name} Age: {self.age}")

tom = Person("Tom", 22)
tom.display_info()# Name: Tom Age: 22
bob = Person("Bob", 43)
bob.display_info()# Name: Bob Age: 43
```

Тут визначається метод display_info(), який виводить інформацію на консоль. І для звернення в методі до атрибутів об'єкта застосовується слово self: `self.name` і `self.age`

У результаті ми отримаємо наступний консольний висновок:

```
Name: Tom Age: 22
Name: Bob Age: 43
```

Деструктори

Крім конструкторів, класи в Python також можуть визначати спеціальні методи - деструктори , які викликаються при видаленні об'єкта. Деструктор є метод __del__(self) , який, як й у конструктор, передається посилання поточний об'єкт. У деструкторі визначаються дії, які треба виконати при видаленні об'єкта, наприклад, звільнення чи видалення ресурсів, які використовував об'єкт.

Деструктор викликається автоматично інтерпретатором, нам не потрібно його явно викликати. Найпростіший приклад:

```
class Person:

    def __init__(self, name):
        self.name = name
        print("Створено людину з ім'ям", self.name)

    def __del__(self):
        print("Вилучена людина з ім'ям", self.name)

tom = Person("Tom")
```

Тут деструктор просто виведе повідомлення про видалення об'єкта Person. Програма створює один об'єкт Person і зберігає посилання на нього у змінній . Створення об'єкта викличе виконання конструктора. При завершенні програми автоматично виконуватиметься деструктор об'єкта Tom. У результаті консольний висновок програми буде таким:

```
Створено людину з ім'ям Tom
Вилучена людина з ім'ям Tom
```

Інший приклад:

```
class Person:

    def __init__(self, name):
        self.name = name
        print("Створено людину з ім'ям", self.name)

    def __del__(self):
        print("Вилучена людина з ім'ям", self.name)

def create_person():
    tom = Person("Tom")

create_person()
print("Кінець програми")
```

Тут об'єкт Person створюється і використовується всередині функції create_person, тому життя об'єкта Person, що створюється, обмежена областю цієї функції. Відповідно, коли функція завершить своє виконання, об'єкт Person викликатиме деструктор. У результаті ми отримаємо наступний консольний висновок:

```
Створено людину з ім'ям Tom
Вилучена людина з ім'ям Tom
```

```
Кінець програми
```

Додаткові матеріали

- Вправи для самоперевірки

Інкапсуляція, атрибути та властивості
За умовчанням атрибути у класах є загальнодоступними, а це означає, що з будь-якого місця програми ми можемо отримати атрибут об'єкта та змінити його. Наприклад:

```
class Person:
    def __init__(self, name, age):
        self.name = name# встановлюємо ім'я
        self.age = age# встановлюємо вік

    def print_person(self):
        print(f"Ім'я: {self.name}\tВік: {self.age}")

tom = Person("Tom", 39)
tom.name = "Людина-павук"# змінюємо атрибут name
tom.age = -129# змінюємо атрибут age
tom.print_person()# Ім'я: Людина-павук Вік: -129
```

Але в даному випадку ми можемо, наприклад, надати віку або імені людини некоректне значення, наприклад, вказати негативний вік. Подібна поведінка небажана, тому постає питання контролю за доступом до атрибутів об'єкта.

З цією проблемою тісно пов'язане поняття інкапсуляції. Інкапсуляція є фундаментальною концепцією об'єктно-орієнтованого програмування, яка передбачає приховування функціоналу та запобігання прямому доступу ззовні до нього.

Мова програмування Python дозволяє визначити приватні або закриті атрибути. Для цього ім'я атрибута повинне починатися з подвійного підкреслення - `__name`. Наприклад, перепишемо попередню програму, зробивши обидва атрибути - name і age приватними:

```
class Person:
    def __init__(self, name, age):
        self.__name = name# встановлюємо ім'я
        self.__age = age# встановлюємо вік

    def print_person(self):
        print(f"Ім'я: {self.__name}\tВік: {self.__age}")
```

```
tom = Person("Tom", 39)
tom.__name = "Людина-павук"# намагаємося змінити атрибут __name
tom.__age = -129# намагаємося змінити атрибут __
tom.print_person()# Ім'я: Tom Вік: 39
```

В принципі, ми також можемо спробувати встановити для атрибутів `__name` і `__age` нові значення:

```
tom.__name = "Людина-павук"# намагаємося змінити атрибут __name
tom.__age = -129# намагаємося змінити атрибут __
```

Але висновок методу print_person покаже, що атрибути об'єкта не змінили значення:

```
tom.print_person()# Ім'я: Tom Вік: 39
```

Як це працює? При оголошенні атрибута, ім'я якого починається з двох прочерків, наприклад, `__attribute`, Python в реальності визначає атрибут, який називається шаблоном `_ClassName__atribute`. Тобто у разі вище створюватимуться атрибути `_Person__name` і `_Person__age`. Тому до такого атрибуту ми зможемо звернутися лише з того самого класу. Але не зможемо звернутися поза цим класом. Наприклад, надання значення цьому атрибуту нічого не дасть:

```
tom.__age = 43
```

Тому що в даному випадку просто визначається динамічно новий атрибут __age, але він не має нічого спільного з атрибутом `self.__age` або точніше `self._Person__age`.

А спроба отримати його значення призведе до помилки виконання (якщо раніше не було визначено змінну __age):

```
print(tom.__age)
```

Проте приватність атрибутів тут досить відносна. Наприклад, ми можемо використати повне ім'я атрибута:

```
class Person:
    def __init__(self, name, age):
        self.__name = name# встановлюємо ім'я
        self.__age = age# встановлюємо вік
```

```
    def print_person(self):
        print(f"Ім'я: {self.__name}\tВік: {self.__age}")


tom = Person("Tom", 39)
tom._Person__name = "Людина-павук"# змінюємо атрибут __name
tom.print_person()# Ім'я: Людина-павук Вік: 39
```

Проте автор зовнішнього коду ще має вгадати, як називаються атрибути.

Методи доступу Геттери та сетери

Може виникнути питання, як звертатися до подібних приватних атрибутів. Для цього зазвичай використовуються спеціальні методи доступу. Геттер дозволяє отримати значення атрибута, а сетер встановити його. Так, змінимо вище певний клас, визначивши у ньому методи доступу:

```
class Person:
    def __init__(self, name, age):
        self.__name = name# встановлюємо ім'я
        self.__age = age# встановлюємо вік
   # Сетер для встановлення віку
    def set_age(self, age):
        if 0 < age < 110:
            self.__age = age
        else:
            print("Неприпустимий вік")
   # Геттер для отримання віку
    def get_age(self):
        return self.__age
   # Геттер для отримання імені
    def get_name(self):
        return self.__name

    def print_person(self):
        print(f"Ім'я: {self.__name}\tВік: {self.__age}")


tom = Person("Tom", 39)
tom.print_person()# Ім'я: Tom Вік: 39
tom.set_age(-3486)# Неприпустимий вік
tom.set_age(25)
tom.print_person()# Ім'я: Tom Вік: 25
```

Для отримання значення віку застосовується метод get_age:

```
def get_age(self):
    return self.__age
```

Для зміни віку визначено метод set_age:

```
def set_age(self, age):
    if 0 < age < 110:
        self.__age = age
    else:
        print("Неприпустимий вік")
```

Причому опосередкування доступу атрибутів через методи дозволяє задати додаткову логіку. Так, залежно від переданого віку ми можемо вирішити, чи треба встановлювати заново вік, оскільки передане значення може бути некоректним.

Також необов'язково створювати для кожного приватного атрибуту подібну пару методів. Так, у прикладі вище ім'я людини ми можемо встановити лише з конструктора. А для отримання визначено метод get_name.

Анотації властивостей

Ми розглянули, як створювати методи доступу. Але Python має ще один - більш елегантний спосіб - властивості . Цей спосіб передбачає використання анотацій, які передуються символом @.

Для створення властивості-геттера над властивістю ставиться інструкція @property .

Для створення властивості-сеттера над властивістю встановлюється інструкція ім'я_властивості_геттера.setter .

Перепишемо клас Person з використанням анотацій:

```
class Person:
    def __init__(self, name, age):
        self.__name = name# встановлюємо ім'я
        self.__age = age# встановлюємо вік
   # властивість-гетер
    @property
    def age(self):
        return self.__age
   # властивість-сетер
    @age.setter
    def age(self, age):
        if 0 < age < 110:
```

```
            self.__age = age
        else:
            print("Неприпустимий вік")
    @property
    def name(self):
        return self.__name

    def print_person(self):
        print(f"Ім'я: {self.__name}\tВік: {self.__age}")


tom = Person("Tom", 39)
tom.print_person()# Ім'я: Tom Вік: 39
tom.age = -3486# Неприпустимий вік (Звернення до сетера)
print(tom.age)# 39 (Звернення до гетера)
tom.age = 25# (Звернення до сетера)
tom.print_person()# Ім'я: Tom Вік: 25
```

По-перше, варто звернути увагу на те, що властивість-сеттер визначається після властивості-геттера.

По-друге, і сетер, і гетер називаються однаково - age. І оскільки геттер називається age, то над сеттером встановлюється інструкція `@age.setter`.

Після того, що до геттера, що до сетера, ми звертаємося через вираз `tom.age`.

При цьому можна визначити лише геттер, як у випадку з властивістю name - його не можна змінити, а можна лише отримати значення.

успадкування
Спадкування дозволяє створювати новий клас на основі вже існуючого класу. Поряд з інкапсуляцією успадкування є одним із наріжних каменів об'єктно-орієнтованого програмування.

Ключовими поняттями успадкування є підклас та суперклас . Підклас успадковує від суперкласу всі публічні атрибути та методи. Суперклас ще називається базовим (base class) чи батьківським (parent class), а підклас – похідним (derived class) чи дочірнім (child class).

Синтаксис для успадкування класів виглядає так:

```
class підклас (суперклас):
    методи_підкласу
```

Наприклад, у нас є клас Person, який представляє людину:

```
class Person:
    def __init__(self, name):
        self.__name = name# ім'я людини
    @property
    def name(self):
        return self.__name

    def display_info(self):
        print(f"Name: {self.__name} ")
```

Припустимо, нам потрібний клас працівника, який працює на деякому підприємстві. Ми могли б створити з нуля новий клас, наприклад, клас Employee:

```
class Employee:
    def __init__(self, name):
        self.__name = name# ім'я працівника
    @property
    def name(self):
        return self.__name
    def display_info(self):
        print(f"Name: {self.__name} ")
    def work(self):
        print(f"{self.name} works")
```

Проте клас Employee може мати самі атрибути і методи, як і клас Person, оскільки працівник - це людина. Так, вище в класі Employee тільки додається метод
`works`, весь інший код повторює функціонал класу Person. Але щоб не дублювати функціонал одного класу в іншому, краще краще застосувати успадкування.

Отже, успадкуємо клас Employee від класу Person:

```
class Person:
    def __init__(self, name):
        self.__name = name# ім'я людини
    @property
    def name(self):
        return self.__name
    def display_info(self):
        print(f"Name: {self.__name} ")

class Employee(Person):
    def work(self):
        print(f"{self.name} works")
```

```
tom = Employee("Tom")
print(tom.name)# Tom
tom.display_info()# Name: Tom
tom.work()# Tom works
```

Клас Employee повністю переймає функціонал класу Person лише додаючи метод `work()`. Відповідно при створенні об'єкта Employee ми можемо використовувати успадкований від Person конструктор:

```
tom = Employee("Tom")
```

І також можна звертатися до успадкованих атрибутів/властивостей та методів:

```
print(tom.name)# Tom
tom.display_info()# Name: Tom
```

Проте варто звернути увагу, що для Employee НЕ доступні закриті атрибути типу __name. Наприклад, ми не можемо в методі work звернутися до приватного атрибуту
`self.__name`:

```
def work(self):
    print(f"{self.__name} works")# ! Помилка
```

Множинне успадкування

Однією з відмінних рис Python є підтримка множинного успадкування, тобто один клас можна успадкувати від декількох класів:

```
клас працівника
class Employee:
    def work(self):
        print("Employee works")

# клас студента
class Student:
    def study(self):
        print("Student studies")

class WorkingStudent(Employee, Student):# Спадкування від класів
Employee та Student
    pass
```

```
# клас працюючого студента
tom = WorkingStudent()
tom.work()# Employee works
tom.study()# Student studies
```

Тут визначено клас Employee, який представляє співробітника фірми, та клас Student, який представляє студента. Клас WorkingStudent, який представляє працюючого студента, не визначає жодного функціоналу, тому в ньому визначено оператора pass . Клас WorkingStudent просто успадковує функціонал від двох класів Employee та Student. Відповідно об'єкт цього класу ми можемо викликати методи обох класів.

При цьому успадковані класи можуть бути більш складними за функціональністю, наприклад:

```
class Employee:
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    def work(self):
        print(f"{self.name} works")

class Student:
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    def study(self):
        print(f"{self.name}studios")

class WorkingStudent(Employee, Student):
    pass

tom = WorkingStudent("Tom")
tom.work()# Tom works
tom.study()# Tom studies
```

Множинне успадкування може здатися зручним, проте воно може призвести до плутанини, якщо обидва успадковані класи містять методи/атрибути з однаковими іменами. Наприклад:

```
class Employee:
    def do(self):
        print("Employee works")
```

```
class Student:
    def do(self):
        print("Student studies")


# class WorkingStudent(Student,Employee):
class WorkingStudent(Employee, Student):
    pass
tom = WorkingStudent()
tom.do()#?
```

Обидва базові класи - Employee і Worker визначають метод do, який виводить різний рядок на консоль. Яку саме з цих реалізацій використовуватиме клас-спадкоємець WorkingStudent? При визначенні класу першим у списку базових класів іде клас Employee

```
class WorkingStudent(Employee, Student)
```

Тому реалізація методу do будуть братися із класу Employee.

Якби ми змінили черговість класів:

```
class WorkingStudent(Student,Employee)
```

то використовувалася б реалізація класу Student

За потреби ми можемо програмним чином переглянути черговість застосування функціоналу базових класів. Для цього застосовується атрибут __mro__ або метод mro() :

```
print(WorkingStudent.__mro__)
print(WorkingStudent.mro())
```

Перевизначення функціоналу базового класу
Минулої статті клас Employee повністю переймав функціонал класу Person:

```
class Person:
    def __init__(self, name):
        self.__name = name# ім'я людини
    @property
    def name(self):
        return self.__name
    def display_info(self):
        print(f"Name: {self.__name} ")
```

```
class Employee(Person):
    def work(self):
        print(f"{self.name} works")
```

Але що якщо ми хочемо щось змінити з цього функціоналу? Наприклад, додати працівникові через конструктор новий атрибут, який буде зберігати компанію, де він працює або змінити реалізацію методу display_info. Python дозволяє перевизначити функціонал базового класу.

Наприклад, змінимо класи в такий спосіб:

```
class Person:
    def __init__(self, name):
        self.__name = name# ім'я людини
    @property
    def name(self):
        return self.__name
    def display_info(self):
        print(f"Name: {self.__name}")

class Employee(Person):
    def __init__(self, name, company):
        super().__init__(name)
        self.company = company
    def display_info(self):
        super().display_info()
        print(f"Company: {self.company}")
    def work(self):
        print(f"{self.name} works")

tom = Employee("Tom", "Microsoft")
tom.display_info()# Name: Tom
                  # Company: Microsoft
```

Тут у класі Employee додається новий атрибут `self.company`, який зберігає компанія працівника. Відповідно метод `__init__()` приймає три параметри: другий для встановлення імені та третій для встановлення компанії. Але якщо в базовому класі визначено конструктора за допомогою методу __init__, і ми хочемо у похідному класі змінити логіку конструктора, то в конструкторі похідного класу ми повинні викликати конструктор базового класу. Тобто, в конструкторі Employee треба викликати конструктор класу Person.

Для звернення до базового класу використовується вираз super() . Так, у конструкторі Employee виконується виклик:

```
super().__init__(name)
```

Цей вираз представлятиме виклик конструктора класу Person, який передається ім'я працівника. І це логічно. Адже ім'я працівника встановлюється у конструкторі класу Person. У самому конструкторі Employee лише встановлюємо якість company.

Крім того, у класі Employee перевизначається метод `display_info()` - до нього додається висновок компанії працівника. Причому ми могли визначити цей метод так:

```
def display_info(self):
    print(f"Name: {self.name}")
    print(f"Company: {self.company}")
```

Але тоді рядок виведення імені повторював би код із класу Person. Якщо ця частина коду збігається з методом класу Person, то немає сенсу повторюватися, тому знову ж таки за допомогою виразу super() звертаємося до реалізації методу display_info в класі Person:

```
def display_info(self):
    super().display_info()# звернення до методу display_info у класі
Person
    print(f"Company: {self.company}")
```

Потім ми можемо викликати конструктор Employee для створення об'єкта цього класу та викликати метод display_info:

```
tom = Employee("Tom", "Microsoft")
tom.display_info()
```

Консольний висновок програми:

```
Name: Tom
Company: Microsoft
```

Перевірка типу об'єкта

p align="justify"> При роботі з об'єктами буває необхідно в залежності від їх типу виконати ті чи інші операції. І за допомогою вбудованої функції isinstance() ми можемо перевірити тип об'єкта. Ця функція приймає два параметри:

```
isinstance(object, type)
```

Перший параметр представляє об'єкт, а другий - тип, належність якого виконується перевірка. Якщо об'єкт представляє вказаний тип, функція повертає True. Наприклад, візьмемо наступну ієрархію класів Person-Employee/Student:

```
class Person:
    def __init__(self, name):
        self.__name = name# ім'я людини
    @property
    def name(self):
        return self.__name
    def do_nothing(self):
        print(f"{self.name} does nothing")

клас працівника
class Employee(Person):
    def work(self):
        print(f"{self.name} works")

# клас студента
class Student(Person):
    def study(self):
        print(f"{self.name}studios")

def act(person):
    if isinstance(person, Student):
        person.study()
    elif isinstance(person, Employee):
        person.work()
    elif isinstance(person, Person):
        person.do_nothing()

tom = Employee("Tom")
bob = Student("Bob")
sam = Person("Sam")
act(tom)# Tom works
act(bob)# Bob studies
act(sam)# Sam does nothing
```

Тут клас Employee визначає метод work(), а клас Student – метод study.

Тут також визначено функцію `act`, яка перевіряє за допомогою функції `isinstance`, чи є параметр person певний тип, і залежно від результатів перевірки звертається до певного методу об'єкта.

Атрибути класів та статичні методи

Атрибути класу

Окрім атрибутів об'єктів у класі можна визначати атрибути класів. Подібні атрибути визначаються як змінних рівня класу. Наприклад:

```
class Person:
     type = "Person"
     description = "Describes a person"

print(Person.type)# Person
print(Person.description)# Describes a person
Person.type = "Class Person"
print(Person.type)# Class Person
```

Тут у класі Person визначено два атрибути: type, який зберігає ім'я класу, та description, який зберігає опис класу.

Для звернення до атрибутів класу ми можемо використовувати ім'я класу, наприклад: `Person.type`, і, як і атрибути об'єкта, ми можемо отримувати та змінювати їх значення.

Подібні атрибути є спільними для всіх класових об'єктів:

```
class Person:
     type = "Person"
     def __init__(self, name):
         self.name = name

tom = Person("Tom")
bob = Person("Bob")
print(tom.type)# Person
print(bob.type)# Person
# змінимо атрибут класу
Person.type = "Class Person"
print(tom.type)# Class Person
print(bob.type)# Class Person
```

Атрибути класу можуть бути використані для таких ситуацій, коли нам потрібно визначити деякі загальні дані для всіх об'єктів. Наприклад:

```
class Person:
    default_name = "Undefined"
    def __init__(self, name):
        if name:
            self.name = name
        else:
            self.name = Person.default_name
```

```
tom = Person("Tom")
bob = Person("")
print(tom.name)# Tom
print(bob.name)# Undefined
```

У цьому випадку атрибут default_name зберігає стандартне ім'я. І якщо конструктор передано порожній рядок для імені, то атрибуту name передається значення атрибута класу default_name. Для звернення до атрибуту класу всередині методів можна використовувати ім'я класу

```
self.name = Person.default_name
```

Атрибут класу

Можлива ситуація, коли атрибут класу та атрибут об'єкта збігаються на ім'я. Якщо код для атрибута об'єкта не задано значення, то для нього може застосовуватися значення атрибута класу:

```
class Person:
    name = "Undefined"
    def print_name(self):
        print(self.name)

tom = Person()
bob = Person()
tom.print_name()# Undefined
bob.print_name()# Undefined
bob.name = "Bob"
bob.print_name()# Bob
tom.print_name()# Undefined
```

Тут метод print_name використовує атрибут об'єкту name, проте ніде в коді цей атрибут не встановлюється. На рівні класу заданий атрибут name. Тому при першому зверненні до методу print_name в ньому буде використовуватися значення атрибута класу:

```
tom = Person()
bob = Person()
tom.print_name()# Undefined
bob.print_name()# Undefined
```

Але далі ми можемо змінити встановити атрибут об'єкта:

```
bob.name = "Bob"
bob.print_name()# Bob
tom.print_name()# Undefined
```

Причому другий об'єкт – tom продовжить використовувати атрибут класу. І якщо ми змінимо атрибут класу, відповідно значення `tom.name` також зміниться:

```
tom = Person()
bob = Person()
tom.print_name()# Undefined
bob.print_name()# Undefined
Person.name = "Some Person"# змінюємо значення атрибута класу
bob.name = "Bob"# встановлюємо атрибут об'єкта
bob.print_name()# Bob
tom.print_name()# Some Person
```

Статичні методи

Крім звичайних методів, клас може визначати статичні методи. Такі методи передуються інструкцією @staticmethod і ставляться загалом до класу. Статичні методи зазвичай визначають поведінку, яка залежить від конкретного об'єкта:

```
class Person:
    __type = "Person"
    @staticmethod
    def print_type():
        print(Person.__type)

Person.print_type()# Person - звернення до статичного методу через
ім'я класу
tom = Person()
tom.print_type()# Person - звернення до статичного методу через ім'я
об'єкта
```

У разі у класі Person визначено атрибут класу `__type`, який зберігає значення, загальне всім класу - назва класу. Причому оскільки назва атрибуту передує двома підкресленнями, то цей атрибут буде приватним, що захистить від неприпустимої зміни.

Також у класі Person визначено статичний метод `print_type`, який виводить на консоль значення атрибуту __type. Дія цього методу не залежить від конкретного об'єкта і відноситься загалом до всього класу - незалежно від об'єкта на консоль буде виводиться одне й те саме значення атрибуту __type. Тому такий метод можна зробити статичним.

Клас об'єкта. Строкове представлення об'єкта
Починаючи з 3-ї версії в мові програмування Python, всі класи неявно мають один загальний суперклас - object і всі класи за замовчуванням успадковують його методи.

Одним із найбільш використовуваних методів класу object є метод __str__() . Коли необхідно отримати рядкове представлення об'єкта або вивести об'єкт у вигляді рядка, Python якраз викликає цей метод. І щодо класу хорошої практикою вважається перевизначення цього.

Наприклад, візьмемо клас Person і виведемо його рядкову виставу:

```
class Person:
    def __init__(self, name, age):
        self.name = name# встановлюємо ім'я
        self.age = age# встановлюємо вік
    def display_info(self):
        print(f"Name: {self.name} Age: {self.age}")

tom = Person("Tom", 23)
print(tom)
```

Під час запуску програма виведе щось на кшталт наступного:

```
<__main__.Person object at 0x10a63dc00>
```

Це не надто інформативна інформація про об'єкт. Ми, звичайно, можемо вийти зі становища, визначивши в класі Person додатковий метод, який виводить дані об'єкта – у прикладі вище це метод display_info.

Але є й інший вихід - визначимо в класі Person метод __str__() (по два підкреслення з кожного боку):

```
class Person:
    def __init__(self, name, age):
        self.name = name# встановлюємо ім'я
        self.age = age# встановлюємо вік
    def display_info(self):
        print(self)
       # print(self.__str__())# або так
    def __str__(self):
        return f"Name: {self.name} Age: {self.age}"

tom = Person("Tom", 23)
```

```
print(tom)# Назви: Tom Age: 23
tom.display_info()# Name: Tom Age: 23
```

Метод `__str__` має повертати рядок. І в цьому випадку ми повертаємо базову інформацію про людину. Якщо нам потрібно використовувати цю інформацію в інших методах класу, ми можемо використовувати вираз `self.__str__()`

І тепер консольний висновок буде іншим:

```
Name: Tom Age: 23
Name: Tom Age: 23
```

Перевантаження операторів
Python дозволяє визначати для своїх класів вбудовані оператори, такі як операції складання, віднімання тощо. Для цього в модулі operator визначено низку функцій:

Операція	**Синтаксис**	**Функція**
Додавання	`a + b`	`__add__(a, b)`
---	---	---
Об'єднання	`seq1 + seq2`	`__concat__(seq1, seq2)`
Перевірка наявності	`obj in seq`	`__contains__(seq, obj)`
Поділ	`a / b`	`__truediv__(a, b)`
Поділ	`a // b`	`__floordiv__(a, b)`
Порозрядне І	`a & b`	`__and__(a, b)`
Порозрядне XOR	`a ^ b`	`__xor__(a, b)`
Порозрядна інверсія	`~ a`	`__invert__(a)`
Порозрядне АБО	`a	`b`
Ступінь	`a ** b`	`__pow__(a, b)`
Присвоєння за індексом	`obj[k] = v`	`__setitem__(obj, k, v)`
Вилучення за індексом	`del obj[k]`	`__delitem__(obj, k)`
Звернення за індексом	`obj[k]`	`__getitem__(obj, k)`
Зрушення вліво	`a << b`	`__lshift__(a, b)`
Залишок від розподілу	`a % b`	`__mod__(a, b)`
множення	`a * b`	`__mul__(a, b)`
Розмноження матриць	`a @ b`	`__matmul__(a, b)`
Арифметичне заперечення	`-a`	`__neg__(a)`
Логічне заперечення	`not a`	`__not__(a)`

Операція	Синтаксис	Функція
Позитивне значення	`+a`	`__pos__(a)`
Зсув праворуч	`a >> b`	`__rshift__(a, b)`
Встановлення діапазону	`seq[i:j] = values`	`__setitem__(seq, slice(i, j), values)`
Видалення діапазону	`del seq[i:j]`	`__delitem__(seq, slice(i, j))`
Отримання діапазону	`seq[i:j]`	`__getitem__(seq, slice(i, j))`
Віднімання	`a - b`	`__sub__(a, b)`
Перевірка на True/False	`obj`	`__bool__(obj)`
Менше ніж	`a < b`	`__lt__(a, b)`
Менше ніж чи одно	`a <= b`	`__le__(a, b)`
Рівність	`a==b`	`__eq__(a, b)`
Нерівність	`a != b`	`__ne__(a, b)`
Більш ніж чи одно	`a >= b`	`__ge__(a, b)`
Більш ніж	`a > b`	`__gt__(a, b)`
Додавання з привласненням	`a += b`	`__iadd__(a, b)`
Об'єднання із присвоєнням	`a += b`	`__iconcat__(a, b)`
Порозрядне множення із присвоєнням	`a &= b`	`__iand__(a, b)`
Поділ із присвоєнням	`a //= b`	`__ifloordiv__(a, b)`
Зсув ліворуч із присвоєнням	`a <<= b`	`__ilshift__(a, b)`
Зсув праворуч із присвоєнням	`a >>= b`	`__irshift__(a, b)`
Розподіл по модулю із присвоєнням	`a %= b`	`__imod__(a, b)`
Множення з присвоєнням	`a += b`	`__imul__(a, b)`
Розмноження матриць із присвоєнням	`a @= b`	`__imatmul__(a, b)`
Порозрядне додавання з привласненням	`a	= b`
Зведення у ступінь із присвоєнням	`a **= b`	`__ipow__(a, b)`
Віднімання з присвоєнням	`a -= b`	`__isub__(a, b)`
Поділ із присвоєнням	`a /= b`	`__itruediv__(a, b)`

Операція	Синтаксис	Функція
Операція XOR із присвоєнням	`a ^= b`	`__ixor__(a, b)`

Щоб визначити оператор для деякого класу, цей клас має реалізувати відповідну функцію. Так, визначення оператора складання застосовується функція __add__() , тому всередині класу нам треба визначити цю функцію. Наприклад:

```
class Counter:
    def __init__(self, value):
        self.value = value
   # Перевизначення оператора додавання
    def __add__(self, other):
        return Counter(self.value + other.value)

counter1 = Counter(5)
counter2 = Counter(15)
counter3 = counter1 + counter2
print(counter3.value)# 20
```

Тут визначено клас Counter, який має атрибут value – умовне деяке число. За допомогою функції `__add__` визначаємо для типу Counter оператор додавання. Допустимо, ми хочемо, щоб один об'єкт Counter можна було скласти з іншим об'єктом Counter. У цьому випадку другий параметр функції представлятиме інший об'єкт Counter:

```
def __add__(self, other):
    return Counter(self.value + other.value)
```

В результаті повертаємо новий об'єкт Counter, в який міститься сума атрибутів значення обох об'єктів.

Після цього ми зможемо складати два об'єкти Counter, і результатом додавання буде новий об'єкт Counter.

Причому в даному випадку реалізовано не єдиний можливий варіант оператора додавання. Так, у прикладі вище другий параметр функції представляв інший об'єкт Counter. Але в реальності це може бути будь-який тип. Наприклад, якщо ми хочемо складати Counter не з іншим об'єктом Counter, а з числом. Тоді ми могли визначити наступний оператор:

```
class Counter:
    def __init__(self, value):
        self.value = value
```

```
    def __add__(self, other):
        return Counter(self.value + other)

counter1 = Counter(5)
counter3 = counter1 + 6
print(counter3.value)# 11
```

Тут оператор додавання як і раніше повертає об'єкт Counter, тільки тепер другий параметр представляє звичайне число.

Тип типу ряду операторів, що повертається, також жорстко не визначений. Наприклад, ми могли б повернути також звичайне число:

```
class Counter:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return self.value + інші

counter1 = Counter(5)
result = counter1 + 7
print(result)# 12
```

Розглянемо ще низку прикладів визначення операторів.

Істинність об'єкта

Визначення функції __bool__ дозволяє встановити істинність об'єкта або фактично перетворити об'єкт на значення True/False. Наприклад:

```
class Counter:
    def __init__(self, value):
        self.value = value
    def __bool__(self):
        return self.value > 0

def test(counter):
    if counter: print("Counter = True")
    else: print("Counter = False")

counter1 = Counter(3)
test(counter1)# Counter = True
counter2 = Counter(-3)
test(counter2)# Counter = False
```

В даному випадку вважатимемо, що, якщо значення value в Counter менше 1, то об'єкт Counter буде розглядатися як False, а при позитивних значеннях - як True. Завдяки цьому ми можемо використовувати об'єкт Counter в умовних чи циклічних конструкціях. Так, у прикладі вище для тестування визначено функцію test, яка в конструкції if..else перевіряє значення об'єкта Counter і в залежності від результату перевірки виводить певне повідомлення на консоль.

Або, наприклад, ми могли б використовувати об'єкт Counter у циклі while як умова:

```
class Counter:
    def __init__(self, value):
        self.value = value
    def __bool__(self):
        return self.value > 0

counter1 = Counter(3)
while(counter1):
    print("Counter1: ", counter1.value)
    counter1.value -=1
```

В даному випадку цикл while буде виконуватися, поки counter1 відповідає значенню True (по суті, його значення value більше 0)

Оператори які повертають значення bool

Ряд операцій покликані повертати логічне значення True чи False . Наприклад, операції порівняння:

```
class Counter:
    def __init__(self, value):
        self.value = value

    def __gt__(self, other):
        return self.value > other.value
    def __lt__(self, other):
        return self.value < other.value


counter1 = Counter(1)
counter2 = Counter(2)

if counter1 > counter2:
    print("counter1 більше ніж counter2")
elif counter1 < counter2:
    print("counter1 менше ніж counter2")
```

```
else:
    print("counter1 та counter2 рівні")
```

Тут у класі Counter визначені оператори <(функція `__lt__()`) та >(функція `__gt__()`). У цьому випадку порівнюємо з іншим об'єктом Counter. Насправді ж порівнюємо значення атрибутів двох об'єктів.

```
def __gt__(self, other):
    return self.value > other.value
def __lt__(self, other):
    return self.value < other.value
```

Потім ми можемо застосовувати відповідні операції до двох об'єктів Counter:

```
if counter1 > counter2:
```

Операції звернення за індексом

Ряд операторів дозволяють звертатися до об'єкта за індексом, використовуючи квадратні дужки:

```
obj[index]
```

Зазвичай подібні операції застосовуються до колекцій, які будуть розглянуті в наступних статтях. Наприклад, можна використовувати подібні операції для отримання або зміни певного елемента списку значень. Насправді ці операції можуть застосовуватися до будь-якого об'єкта, а індекс, що використовується, також може представляти все що завгодно. Розглянемо операції звернення за індексом на прикладі отримання значення за індексом:

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def __getitem__(self, prop):
        if prop == "name": return self.__name
        elif prop == "age": return self.__age
        return None

tom = Person("Tom", 39)
print("Name:", tom["name"])# Name: Tom
print("Age:", tom["age"])# Age: 39
print("Id:", tom["id"])# Id: None
```

Отже, тут визначено клас Person, що містить два приватні поля - __name і __age. Для реалізації отримання даних за індексом визначено функцію __getitem__() . Як другий параметр цю функцію передається значення, яке виконує роль індексу. У нашому випадку це буде назва атрибуту. І залежно від переданого значення повертаємо або значення атрибуту __name, або значення атрибуту __age. Якщо передано невалідну назву атрибута, то повертаємо None.

Для отримання значення по индес передаємо індекс - назва атрибута в квадратних дужках:

```
tom["name"]
```

Перевірка наявності якості

Оператор in дозволяє перевірити наявність певного значення послідовності - деякому наборі значень:

```
значення in послідовність
```

Якщо значення є у послідовності, то повертається True, інакше повертається False. Наприклад, перевіримо наявність властивості в об'єкті:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __contains__(self, prop):
        if prop == "name" or prop == "age": return True
        return False

tom = Person("Tom", 39)
print("name" in tom)# True
print("id" in tom)# False
```

За реалізацію оператора in відповідає функція __contains__() . Як перший параметр, як завжди, вказується поточний об'єкт - той об'єкт, який стоїть праворуч від оператора in . А як другий параметр - перевірене значення - воно вказується зліва від in . У разі якщо другий параметр - дорівнює " name " чи " age " , то повертаємо True. Що означає, що атрибут є в об'єкті?

Відповідно вираз `"name" in tom` поверне True, а вираз `"id" in tom` поверне False.

Реалізація операторів парами

Деякі оператори – оператори порівняння зручніше реалізувати парами. Якщо ми реалізуємо оператор == , можна відразу реалізувати і оператор != . Причому щоб не прописувати ту саму логіку по два рази, можна реалізувати один оператор через інший:

```
class Counter:
    def __init__(self, value):
        self.value = value

    def __eq__(self, other): return self.value == other.value
    def __ne__(self, other): return not (self == other)

    def __gt__(self, other): return self.value > other.value
    def __le__(self, other): return not (self > other)

    def __lt__(self, other): return self.value < other.value
    def __ge__(self, other): return not (self < other)

c1 = Counter(1)
c2 = Counter(2)
print(c1 == c2)# False
print(c1 != c2)# True
print(c1 < c2)# True
print(c1 >= c2)# False
```

У цьому випадку оператора != повертає інверсію результату оператора == , який визначено вище

```
def __eq__(self, other): return self.value == other.value
def __ne__(self, other): return not (self == other)
```

Аналогічно визначені оператори < і >=, і навіть > і <=.

Додаткові матеріали

- Вправи для самоперевірки

Розділ 4. Обробка помилок та винятків

Конструкція try...except...finally

При програмуванні Python ми можемо зіткнутися з двома типами помилок. Перший тип представляють синтаксичні помилки (syntax error). Вони з'являються внаслідок порушення синтаксису мови програмування під час написання вихідного коду. При роботі в будь-якому середовищі розробки, наприклад,

PyCharm, IDE сама може відстежувати синтаксичні помилки і яким-небудь чином їх виділяти.

Другий тип помилок є помилками виконання (runtime error). Вони з'являються вже в процесі виконання програми. Подібні помилки ще називають винятками. Наприклад, у минулих темах ми розглядали перетворення рядка на число:

```
string = "5"
number = int(string)
print(number)
```

Цей скрипт успішно виконається, тому що рядок "5" цілком може бути конвертований у число. Однак візьмемо інший приклад:

```
string = "hello"
number = int(string)
print(number)
```

При виконанні цього скрипту буде викинуто виключення ValueError, оскільки рядок "hello" не можна перетворити на число:

```
ValueError: invalid literal for int() with base 10: 'hello'
```

З одного боку, тут очевидно, що рядок не є числом, але ми можемо мати справу з введенням користувача, який також може ввести не зовсім те, що ми очікуємо:

```
string = input("Введіть число: ")
number = int(string)
print(number)
```

При виникненні виключення робота програми переривається, і щоб уникнути подібної поведінки та обробляти винятки в Python є конструкція try..except .

try..except

Конструкція try..except має таке формальне визначення:

```
try:
    інструкції
except [Тип_виключення]:
    інструкції
```

Весь основний код, у якому потенційно може виникнути виняток, міститься після ключового слова try . Якщо в цьому коді генерується виняток, робота коду в блоці try переривається, і виконання переходить в блок except .

Після ключового слова except опціонально можна вказати, який виняток буде оброблятись (наприклад, ValueError або KeyError). Після слова except на наступному стоку йдуть інструкції блоку except, що виконуються у разі виключення.

Розглянемо обробку виключення на прикладі перетворення рядка в число:

```
try:
    number = int(input("Введіть число: "))
    print("Введене число:", number)
except:
    print("Перетворення пройшло невдало")
print("Завершення програми")
```

Вводимо рядок:

```
Введіть число: hello
Перетворення пройшло невдало
Завершення програми
```

Як видно з консольного виведення, при введенні рядка виведення числа на консоль не відбувається, а виконання програми переходить до блоку except.

Вводимо правильне число:

```
Введіть число: 22
Введене число: 22
Завершення програми
```

Тепер все виконується нормально, виняток не виникає, і відповідно блок except не виконується.

Блок finally

При обробці винятків можна використовувати необов'язковий блок finally . Відмінною особливістю цього блоку є те, що він виконується незалежно, чи було згенеровано виняток:

```
try:
    number = int(input("Введіть число: "))
    print("Введене число:", number)
```

```
except:
    print("Перетворення пройшло невдало")
finally:
    print("Блок try завершив виконання")
print("Завершення програми")
```

Як правило, блок finally застосовується для звільнення ресурсів, що використовуються, наприклад, для закриття файлів.

Варто зазначити, що блок finally не обробляє винятки, і якщо ми використовуємо цей блок без блоку except , то при виникненні помилки додаток аварійно завершиться, як у наступному випадку при розподілі числа на нуль:

```
try:
    number = 3/0# генерує виняток ZeroDivisionError
    print(number)
finally:
    print("Блок try завершив виконання")
print("Завершення програми")
```

Додаткові матеріали

- Запитання для самоперевірки

except та обробка різних типів винятків

Вбудовані типи винятків

У прикладі вище оброблялися відразу всі винятки, які можуть виникнути у коді. Однак ми можемо конкретизувати тип виключення, що обробляється, вказавши його після слова except:

```
try:
    number = int(input("Введіть число: "))
    print("Введене число:", number)
except ValueError:
    print("Перетворення пройшло невдало")
print("Завершення програми")
```

У цьому випадку блок execpt обробляє лише винятки типу ValueError , які можуть виникнути при невдалому перетворенні рядка на число.

У Python є такі базові типи винятків:

- BaseException : базовий тип для всіх вбудованих винятків
- Exception : базовий тип, який зазвичай застосовується для створення своїх типів винятків

- ArithmeticError : базовий тип для винятків, пов'язаних з арифметичними операціями (OverflowError, ZeroDivisionError, FloatingPointError).
- BufferError : тип виключення, що виникає за неможливості виконати операцію з буфером
- LookupError : базовий тип для винятків, які виникають при зверненні в колекціях за некоректним ключем або індексом (наприклад, IndexError, KeyError)

Від цих класів успадковуються всі типи винятків. У Python має досить великий список вбудованих винятків. Весь цей список можна переглянути в документації . Перелічу тільки деякі:

- IndexError : виняток виникає, якщо індекс при зверненні до елемента колекції знаходиться поза допустимим діапазоном
- KeyError : виникає, якщо у словнику немає ключа, за яким відбувається звернення до елемента словника.
- OverflowError : виникає, якщо результат арифметичної операції може бути представлений поточним числовим типом (зазвичай типом float).
- RecursionError : виникає, якщо перевищено допустиму глибину рекурсії.
- TypeError : якщо операція або функція застосовується до значення неприпустимого типу.
- ValueError : виникає, якщо операція або функція одержують об'єкт коректного типу з некоректним значенням.
- ZeroDivisionError : виникає при розподілі на нуль.
- NotImplementedError : тип виключення для вказівки, що якісь методи класу не реалізовані
- ModuleNotFoundError : виникає при неможливості знайти модуль при його імпорті директивою `import`
- OSError : тип винятків, які генеруються у разі виникнення помилок системи (наприклад, неможливо знайти файл, пам'ять диска заповнена і т.д.)

І якщо ситуація така, що в програмі можуть бути згенеровані різні типи винятків, ми можемо їх обробити окремо, використовуючи додаткові вирази except. І при виникненні виключення Python шукатиме потрібний блок except, який обробляє цей тип виключення:

```
try:
    number1 = int(input("Введіть перше число: "))
    number2 = int(input("Введіть друге число: "))
    print("Результат поділу:", number1/number2)
except ValueError:
    print("Перетворення пройшло невдало")
except ZeroDivisionError:
    print("Спроба поділу числа на нуль")
```

```
except BaseException:
    print("Загальний виняток")
print("Завершення програми")
```

Якщо виникне виняток у результаті перетворення рядка в число, воно буде оброблено блоком `except ValueError`. Якщо ж друге число дорівнюватиме нулю, тобто буде розподіл на нуль, тоді виникне виняток ZeroDivisionError, і воно буде оброблено блоком `except ZeroDivisionError`.

Тип BaseException є загальним винятком, під який потрапляють всі виняткові ситуації. Тому в даному випадку будь-який виняток, який не представляє тип ValueError або ZeroDivisionError, буде оброблено в блоці `except BaseException:`.

Однак, якщо у програмі виникає виняток типу, для якого немає відповідного блоку `except`, то програма не зможе знайти відповідний блок except і згенерує виняток. Наприклад, у наступному випадку:

```
try:
    number1 = int(input("Введіть перше число: "))
    number2 = int(input("Введіть друге число: "))
    print("Результат поділу:", number1/number2)
except ZeroDivisionError:
    print("Спроба поділу числа на нуль")
print("Завершення програми")
```

Тут передбачена обробка поділу на нуль за допомогою блоку `except ZeroDivisionError`. Однак якщо користувач замість числа введе некорвертіурему число в рядок, то виникне виключення типу ValueError, для якого немає відповідного блоку except. І тому програма аварійно завершить виконання.

Python дозволяє в одному блоці except обробляти відразу кілька типів винятків. В цьому випадку всі типи виключення передаються в дужках:

```
try:
    number1 = int(input("Введіть перше число: "))
    number2 = int(input("Введіть друге число: "))
    print("Результат поділу:", number1/number2)
except (ZeroDivisionError, ValueError):# обробка двох типів винятків
- ZeroDivisionError та ValueError
    print("Спроба поділу числа на нуль або некоректне введення")
print("Завершення програми")
```

Отримання інформації про виключення

За допомогою оператора as ми можемо передати всю інформацію про виключення в змінну, яку можна використовувати в блоці except:

```
try:
    number = int(input("Введіть число: "))
    print("Введене число:", number)
except ValueError as e:
    print("Відомості про виключення", e)
print("Завершення програми")
```

Приклад некоректного введення:

```
Введіть число: fdsf
Відомості про виключення invalid literal for int() with base 10:
'fdsf'
Завершення програми
```

Генерація винятків та створення своїх типів винятків

Генерація винятків та оператор raise

Іноді виникає необхідність вручну згенерувати той чи інший виняток. Для цього застосовується оператор raise . Наприклад, згенеруємо виняток

```
try:
    age = int(input("Введіть вік: "))
    if age > 110 or age < 1:
        raise Exception("Некоректний вік")
    print("Ваш вік:", age)
except ValueError:
    print("Введені некоректні дані")
except Exception as e:
    print(e)
print("Завершення програми")
```

Оператору raise передається об'єкт BaseException - у разі об'єкт Exception. У конструктор цього можна йому передати повідомлення, яке потім можна вивести користувачеві. У результаті, якщо age буде більше 110 або менше 1, то спрацює оператор raise, який згенерує виняток. У результаті керування програмою перейде до блоку except , який обробляє винятки типу Exception:

```
Введіть вік: 100500
Некоректний вік
Завершення програми
```

Створення своїх типів винятків

У мові Python ми не обмежені лише вбудованими типами винятків і можемо, застосовуючи успадкування, у разі потреби створювати свої типи винятків. Наприклад, візьмемо наступний клас Person:

```
class Person:
    def __init__(self, name, age):
        self.__name = name# встановлюємо ім'я
        self.__age = age# встановлюємо вік
    def display_info(self):
        print(f"Ім'я: {self.__name} Вік: {self.__age}")
```

Тут клас Person у конструкторі отримує значення для імені та віку та присвоює їх приватним змінним name та age. Однак при створенні об'єкта Person ми можемо передати в конструктор некоректне з погляду логіки значення, наприклад, негативне число. Одним із способів вирішення даної ситуації є генерація виключення при передачі некоректних значень.

Отже, визначимо наступний код програми:

```
class PersonAgeException(Exception):
    def __init__(self, age, minage, maxage):
        self.age = age
        self.minage = minage
        self.maxage = maxage
    def __str__(self):
        return f"Неприпустиме значення: {self.age}. " \
               f"Вік має бути в діапазоні від {self.minage} до
{self.maxage}"

class Person:
    def __init__(self, name, age):
        self.__name = name# встановлюємо ім'я
        minage, maxage = 1, 110
        if minage < age < maxage:# встановлюємо вік, якщо передано
коректне значення
            self.__age = age
        else:# інакше генеруємо виняток
            raise PersonAgeException(age, minage, maxage)
    def display_info(self):
        print(f"Ім'я: {self.__name} Вік: {self.__age}")
try:
    tom = Person("Tom", 37)
    tom.display_info()# Ім'я: Tom Вік: 37
    bob = Person("Bob", -23)
    bob.display_info()
```

```
except PersonAgeException as e:
    print(e)# Неприпустиме значення: -23. Вік має бути в діапазоні
від 1 до 110
```

Спочатку тут визначено клас PersonAgeException, який успадковується від класу Exception. Зазвичай, власні класи винятків успадковуються від класу Exception. Клас PersonAgeException призначений для винятків, пов'язаних із віком користувача.

У конструкторі PersonAgeException отримуємо три значення - власне некоректне значення, яке спричинило виключення, а також мінімальне та максимальне значення віку.

```
class PersonAgeException(Exception):
    def __init__(self, age, minage, maxage):
        self.age = age
        self.minage = minage
        self.maxage = maxage
    def __str__(self):
        return f"Неприпустиме значення: {self.age}. " \
               f"Вік має бути в діапазоні від {self.minage} до
{self.maxage}"
```

У функції __str__ визначаємо текстове уявлення класу - насправді повідомлення про помилку.

У конструкторі класу Persoon перевіряємо передане віку користувача значення. І якщо це значення не відповідає певному діапазону, то генеруємо виняток типу PersonAgeException:

```
raise PersonAgeException(age, minage, maxage)
```

При застосуванні класу Person слід враховувати, що конструктор класу може згенерувати виняток при передачі некоректного значення. Тому створення об'єктів Person обгортається в конструкцію try.

```
try:
    tom = Person("Tom", 37)
    tom.display_info()# Ім'я: Tom Вік: 37
    bob = Person("Bob", -23)# генерується виняток типу
PersonAgeException
    bob.display_info()
except PersonAgeException as e:
    print(e)# Неприпустиме значення: -23. Вік має бути в діапазоні
від 1 до 110
```

І якщо при виклику конструктора Person буде згенеровано виняток типу PersonAgeException, то керування програмою перейде до блоку except, який обробляє винятки типу PersonAgeException як виведення інформації про виключення на консоль.

Розділ 5. Списки, кортежі та словники

Список

Для роботи з наборами даних Python надає такі вбудовані типи, як списки, кортежі та словники.

Список (list) є типом даних, який зберігає набір або послідовність елементів. Багато мовами програмування є аналогічна структура даних, яка називається масив.

Створення списку

Для створення списку застосовуються квадратні дужки [] , всередині яких через кому перераховуються елементи списку. Наприклад, визначимо список чисел:

```
numbers = [1, 2, 3, 4, 5]
```

Подібним чином можна визначати списки даних інших типів, наприклад, визначимо список рядків:

```
people = ["Tom", "Sam", "Bob"]
```

Також для створення списку можна використовувати функцію-конструктор list() :

```
numbers1 = []
numbers2 = list()
```

Обидва визначення списку аналогічні - вони створюють порожній список.

Список необов'язково має містити лише однотипні об'єкти. Ми можемо помістити в той самий список одночасно рядки, числа, об'єкти інших типів даних:

```
objects = [1, 2.6, "Hello", True]
```

Для перевірки елементів списку можна використовувати стандартну функцію print, яка виводить вміст списку в зручному для читання вигляді:

```
numbers = [1, 2, 3, 4, 5]
people = ["Tom", "Sam", "Bob"]
```

```
print(numbers)# [1, 2, 3, 4, 5]
print(people)# ["Tom", "Sam", "Bob"]
```

Конструктор list може приймати набір значень, на основі яких створюється список:

```
numbers1 = [1, 2, 3, 4, 5]
numbers2 = list(numbers1)
print(numbers2)# [1, 2, 3, 4, 5]
letters = list("Hello")
print(letters)# ['H', 'e', 'l', 'l', 'o']
```

Якщо необхідно створити список, в якому повторюється те саме значення кілька разів, то можна використовувати символ зірочки *, тобто фактично застосувати операцію множення до вже існуючого списку:

```
numbers = [5] * 6# 6 разів повторюємо 5
print(numbers)# [5, 5, 5, 5, 5, 5]
people = ["Tom"] * 3# 3 рази повторюємо "Tom"
print(people)# ["Tom", "Tom", "Tom"]
students = ["Bob", "Sam"] * 2# 2 рази повторюємо "Bob", "Sam"
print(students)# ["Bob", "Sam", "Bob", "Sam"]
```

Звернення до елементів списку

Для звернення до елементів списку треба використовувати індекси, які представляють номер елемента списку. Індекси починаються з нуля. Тобто перший елемент матиме індекс 0, другий елемент індекс 1 і так далі. Для звернення до елементів з кінця можна використовувати негативні індекси, починаючи з -1. Тобто, у останнього елемента буде індекс -1, у передостаннього - -2 і так далі.

```
people = ["Tom", "Sam", "Bob"]
отримання елементів з початку списку
print(people[0])# Tom
print(people[1])# Sam
print(people[2])# Bob
# Отримання елементів з кінця списку
print(people[-2])# Sam
print(people[-1])# Bob
print(people[-3])# Tom
```

Для зміни елемента списку достатньо надати йому нове значення:

```
people = ["Tom", "Sam", "Bob"]
people[1] = "Mike"# зміна другого елемента
print(people[1])# Mike
print(people)# ["Tom", "Mike", "Bob"]
```

Розкладання списку

Python дозволяє розкласти список на окремі елементи:

```
people = ["Tom", "Bob", "Sam"]
tom, bob, sam = люди
print(tom)# Tom
print(bob)# Bob
print(sam)# Sam
```

У разі змінним tom, bob і sam послідовно присвоюються елементи зі списку people. Однак слід враховувати, що кількість змінних повинна дорівнювати числу елементів списку, що присвоюється.

Перебір елементів

Для перебору елементів можна використовувати як цикл for, і цикл while.

Перебір за допомогою циклу for :

```
people = ["Tom", "Sam", "Bob"]
for person in people:
    print(person)
```

Тут буде проводитися перебір списку людей, і кожен його елемент буде поміщатися в змінну людину.

Перебір також можна зробити за допомогою циклу while :

```
people = ["Tom", "Sam", "Bob"]
i = 0
while i < len(people):
    print(people[i])# застосовуємо індекс для отримання елемента
    i += 1
```

Для перебору з допомогою функції len() отримуємо довжину списку. За допомогою лічильника і виводить по елементу, поки значення лічильника не дорівнює довжині списку.

Порівняння списків

Два списки вважаються рівними, якщо вони містять один і той же набір елементів:

```
numbers1 = [1, 2, 3, 4, 5]
numbers2 = list([1, 2, 3, 4, 5])
if numbers1 == numbers2:
    print("numbers1 equal to numbers2")
else:
    print("numbers1 is not equal to numbers2")
```

В даному випадку обидва списки будуть рівними.

Отримання частини списку

Якщо необхідно отримати певну частину списку, то ми можемо застосовувати спеціальний синтаксис, який може приймати такі форми:

- `list[:end]` : через параметр end передається індекс елемента, до якого потрібно скопіювати список
- `list[start:end]` : параметр start вказує на індекс елемента, з якого потрібно скопіювати елементи
- `list[start:end:step]` : параметр step вказує на крок, через який копіюватимуться елементи зі списку. За промовчанням цей параметр дорівнює 1.

```
people = ["Tom", "Bob", "Alice", "Sam", "Tim", "Bill"]
slice_people1 = people[:3]# з 0 по 3
print(slice_people1)# ["Tom", "Bob", "Alice"]
slice_people2 = people[1:3]# з 1 по 3
print(slice_people2)# ["Bob", "Alice"]
slice_people3 = people[1:6:2]# з 1 по 6 з кроком 2
print(slice_people3)# ["Bob", "Sam", "Bill"]
```

Можна використовувати негативні індекси, тоді відлік йтиме з кінця, наприклад, -1 - передостанній, -2 - третій скінця і так далі.

```
people = ["Tom", "Bob", "Alice", "Sam", "Tim", "Bill"]
slice_people1 = people[:-1]# з передостаннього по нульовий
print(slice_people1)# ["Tom", "Bob", "Alice", "Sam", "Tim"]
slice_people2 = people[-3:-1]# з третього з кінця до передостаннього
print(slice_people2)# [ "Sam", "Tim"]
```

Методи та функції по роботі зі списками

Для керування елементами списки мають низку методів. Деякі з них:

- append(item) : додає елемент item до кінця списку
- insert(index, item) : додає елемент item до списку індексу index
- extend(items) : додає набір елементів items до кінця списку
- remove(item) : видаляє елемент item. Видаляється лише перше входження елемента. Якщо елемент не знайдено, генерує виняток ValueError
- clear() : видалення всіх елементів зі списку
- index(item) : повертає індекс елемента item. Якщо елемент не знайдено, генерує виняток ValueError
- pop([index]) : видаляє та повертає елемент за індексом index. Якщо індекс не передано, просто видаляє останній елемент.
- count(item) : повертає кількість входжень елемента item до списку
- sort([key]) : сортує елементи. За умовчанням сортує за зростанням. Але за допомогою key ми можемо передати функцію сортування.
- reverse() : розставляє всі елементи у списку у зворотному порядку
- copy() : копіює список

Крім того, Python надає ряд вбудованих функцій для роботи зі списками:

- len(list) : повертає довжину списку
- sorted(list, [key]) : повертає відсортований список
- min(list) : повертає найменший елемент списку
- max(list) : повертає найбільший елемент списку

Додавання та видалення елементів

Для додавання елемента застосовуються методи , `append()` а для видалення - методи ,
і . `extend``insert``remove()``pop()``clear()`

Використання методів:

```
people = ["Tom", "Bob"]
# додаємо в кінець списку
people.append("Alice")# ["Tom", "Bob", "Alice"]
# додаємо на другу позицію
people.insert(1, "Bill")# ["Tom", "Bill", "Bob", "Alice"]
# додаємо набір елементів ["Mike", "Sam"]
people.extend(["Mike", "Sam"])# ["Tom", "Bill", "Bob", "Alice",
"Mike", "Sam"]
# отримуємо індекс елемента
```

```
index_of_tom = people.index("Tom")
видаляємо за цим індексом
removed_item = people.pop(index_of_tom)# ["Bill", "Bob", "Alice",
"Mike", "Sam"]
видаляємо останній елемент
last_item = people.pop()# ["Bill", "Bob", "Alice", "Mike"]
# видаляємо елемент "Alice"
people.remove("Alice")# ["Bill", "Bob", "Mike"]
print(people)# ["Bill", "Bob", "Mike"]
видаляємо всі елементи
people.clear()
print(people)# []
```

Перевірка наявності елемента

Якщо певний елемент не знайдено, то методи remove та index генерують виняток. Щоб уникнути подібної ситуації, перед операцією з елементом можна перевіряти наявність за допомогою ключового слова in :

```
people = ["Tom", "Bob", "Alice", "Sam"]
if "Alice" in people:
    people.remove("Alice")
print(people)# ["Tom", "Bob", "Sam"]
```

Вираз `if "Alice" in people` повертає True, якщо елемент "Alice" є у списку людей. Тому конструкція `if "Alice" in people`
може виконати наступний блок інструкцій, залежно від наявності елемента у списку.

Видалення за допомогою del

Python також підтримує ще один спосіб видалення елементів списку – за допомогою оператора del . Як параметр цього оператора передається елемент, що видаляється, або набір елементів:

```
people = ["Tom", "Bob", "Alice", "Sam", "Bill", "Kate", "Mike"]
del people[1]# видаляємо другий елемент
print(people)# ["Tom", "Alice", "Sam", "Bill", "Kate", "Mike"]
del people[:3]# видаляємо по четвертий елемент не включаючи
print(people)# ["Bill", "Kate", "Mike"]
del people[1:]# видаляємо з другого елемента
print(people)# ["Bill"]
```

Зміна підписку

Для зміни підписка-набору елементів у списку можна використовувати вищезазначений синтаксис `[start:end]`:

```
nums = [10, 20, 30, 40, 50]
nums[1:4]=[11, 22]
print(nums)# [10, 11, 22, 50]
```

Тут вираз `nums[1:4]` фактично звертається до підпису `[20, 30, 40]`. Присвоєння цьому підписку списку `[11, 22]` дозволяє замінити елемента з 1 по 4 індекс, не включаючи на елементи `[11, 22]`. І після зміни отримаємо список `[10, 11, 22, 50]`

Підрахунок входження

Якщо необхідно дізнатися, скільки разів у списку присутній той чи інший елемент, можна застосувати метод `count()`:

```
people = ["Tom", "Bob", "Alice", "Tom", "Bill", "Tom"]
people_count = people.count("Tom")
print(people_count)# 3
```

Сортування

Для сортування за зростанням застосовується метод sort() :

```
people = ["Tom", "Bob", "Alice", "Sam", "Bill"]
people.sort()
print(people)# ["Alice", "Bill", "Bob", "Sam", "Tom"]
```

Якщо необхідно відсортувати дані у зворотному порядку, то ми можемо після сортування застосувати метод `reverse()`:

```
people = ["Tom", "Bob", "Alice", "Sam", "Bill"]
people.sort()
people.reverse()
print(people)# ["Tom", "Sam", "Bob", "Bill", "Alice"]
```

При сортуванні фактично порівнюються два об'єкти, і який з них "менший", ставиться перед тим, що "більше". Поняття "більше" і "менше" досить умовні. І якщо для чисел все просто - числа розставляють у порядку зростання, то для рядків та інших об'єктів ситуація складніша. Зокрема, рядки оцінюються за

першими символами. Якщо перші символи дорівнюють, оцінюються другі символи тощо. При чому цифровий символ вважається "меншим", ніж алфавітний заголовний символ, а заголовний символ вважається меншим, ніж малий.

Таким чином, якщо у списку поєднуються рядки з верхнім та нижнім регістром, то ми можемо отримати не зовсім коректні результати, тому що для нас рядок "bob" повинен стояти до рядка "Tom". І щоб змінити стандартну поведінку сортування, ми можемо передати в метод `sort()` як параметр функцію:

```
people = ["Tom", "bob", "alice", "Sam", "Bill"]
people.sort()# стандартне сортування
print(people)# ["Bill", "Sam", "Tom", "alice", "bob"]
people.sort(key=str.lower)# сортування без урахування регістру
print(people)# ["alice", "Bill", "bob", "Sam", "Tom"]
```

Крім методу sort, ми можемо використовувати вбудовану функцію sorted , яка має дві форми:

- `sorted(list)` : сортує список list
- `sorted(list, key)` : сортує список list, застосовуючи до елементів функцію key

```
people = ["Tom", "bob", "alice", "Sam", "Bill"]
sorted_people = sorted(people, key=str.lower)
print(sorted_people)# ["alice", "Bill", "bob", "Sam", "Tom"]
```

При використанні цієї функції слід враховувати, що ця функція не змінює список, що сортується, а всі відсортовані елементи вона поміщає в новий список, який повертається як результат.

Фільтрування списку

Для фільтрації списку застосовується функція filter() , в яку передається функція-умова та список, що фільтрується:

```
filter(fun, iter)
```

Функція приймає два параметри:

- `fun` : функція-умова, в яку передається кожен елемент колекції та яка повертає True, якщо елемент відповідає умові. Інакше повертається False.
- `iter` : колекція, що фільтрується.

Як результат, функція повертає відфільтровані елементи. Наприклад, отримаємо зі списку чисел усі значення більше 1:

```
numbers = [-5, -4, -3 ,-2, -1, 0, 1, 2, 3, 4, 5]
def condition(number): return number > 1
result = filter(condition, numbers)
для результатів: print(n, end=" ")# 2 3 4 5
```

Тут фільтрується список номерів. Для фільтрації визначаємо функцію condition, яку як параметра передається кожен елемент списку numbers. Результатом функції є `True`, якщо число більше 1, або `False` якщо число менше 2.

Результатом функції filter є відфільтровані значення зі списку, тобто числа, які більше 1.

Замість визначення окремої функції-умови, якщо умова коротка, зручно в подібних випадках використовувати лямбда-вирази:

```
numbers = [-5, -4, -3 ,-2, -1, 0, 1, 2, 3, 4, 5]
result = filter(lambda n: n > 1, numbers)
для результатів: print(n, end=" ")# 2 3 4 5
```

Аналогічним чином можна відфільтрувати списки складніших об'єктів:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

люди = [Person("Tom", 38), Person("Kate", 31), Person("Bob", 42),
        Person("Alice", 34), Person("Sam", 25)]
# Фільтрування елементів, у яких age > 33
view = filter(lambda p: p.age > 33, people)

for person in view:
    print(f"Name: {person.name} Age: {person.age}")
```

У цьому випадку фільтруємо список об'єктів Person, тому в функцію-умова/лямбда-вираз як параметр передається кожен об'єкт Person зі списку. Кожен об'єкт Person зберігає ім'я (Name) та вік (Age), і тут вибираємо всіх Person, у яких вік більше 33.

Проекція списку

Для проекції/перетворення елементів списку застосовується функція map() , на яку передається функція-умова і список, елементи якого треба перетворити:

```
map(fun, iter)
```

Функція приймає два параметри:

- `fun` : функція перетворення, яку передається кожен елемент колекції.
- `iter` : колекція, що перебирається

З результату функції отримаємо перетворені елементи списку. Наприклад, перетворимо список чисел у квадрати цих чисел:

```
numbers = [1, 2, 3, 4, 5]
def square(number): return number * number
result = map(square, numbers)
для результатів: print(n, end=" ")# 1 4 9 16 25
```

Як функцію перетворення тут виступає функція square, до якої передається число зі списку і яка повертає його квадрат.

Також як функцію перетворення можна використовувати лямбда-вирази:

```
numbers = [1, 2, 3, 4, 5]
result = map(lambda n: n * n, numbers)
для результатів: print(n, end=" ")# 1 4 9 16 25
```

Аналогічним чином можна перетворювати колекції складніших об'єктів:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

люди = [Person("Tom", 38), Person("Kate", 31), Person("Bob", 42),
        Person("Alice", 34), Person("Sam", 25)]
# отримуємо з Person рядок з ім'ям
view = map(lambda p: p.name, people)

for person in view:
    print(person)
```

Тут проекція застосовується до переліку об'єктів Person. Функція перетворення отримує кожен об'єкт Person і повертає значення атрибута name. Тобто ми

отримаємо набір рядків (атрибути name всіх об'єктів Person). Консольний висновок:

```
Tom
Kate
Bob
Alice
Sam
```

Мінімальне та максимальне значення

Вбудований функції Python `min()` і `max()` дозволяють знайти мінімальне та максимальне значення відповідно:

```
numbers = [9, 21, 12, 1, 3, 15, 18]
print(min(numbers))# 1
print(max(numbers))# 21
```

Копіювання списків

При копіюванні списків слід враховувати, що списки представляють тип, що змінюється (mutable), тому якщо обидві змінних будуть вказувати на один і той же список, то зміна однієї змінної, торкнеться і іншу змінну:

```
people1 = ["Tom", "Bob", "Alice"]
people2 = people1
people2.append("Sam")# додаємо елемент до другого списку
# people1 і people2 вказують на той самий список
print(people1)# ["Tom", "Bob", "Alice", "Sam"]
print(people2)# ["Tom", "Bob", "Alice", "Sam"]
```

І щоб відбувалося копіювання елементів, але при цьому змінні вказували на різні списки можна використовувати метод copy() :

```
people1 = ["Tom", "Bob", "Alice"]
people2 = people1.copy()# копіюємо елементи з people1 до people2
people2.append("Sam")# додаємо елемент ТІЛЬКИ до другого списку
# people1 та people2 вказують на різні списки
print(people1)# ["Tom", "Bob", "Alice"]
print(people2)# ["Tom", "Bob", "Alice", "Sam"]
```

З'єднання списків

Для об'єднання списків застосовується операція додавання (+):

```
people1 = ["Tom", "Bob", "Alice"]
people2 = ["Tom", "Sam", "Tim", "Bill"]
people3 = people1 + people2
print(people3)# ["Tom", "Bob", "Alice", "Tom", "Sam", "Tim", "Bill"]
```

Списки списків

Списки крім стандартних даних типу рядків, чисел також можуть містити інші списки. Подібні списки можна порівнювати з таблицями, де вкладені списки виконують роль рядків. Наприклад:

```
people = [
    ["Tom", 29],
    ["Alice", 33],
    ["Bob", 27]
]
print(people[0])# ["Tom", 29]
print(people[0][0])# Tom
print(people[0][1])# 29
```

Щоб звернутися до елемента вкладеного списку, необхідно використати кілька індексів: `people[0][1]` - звернення до другого елемента першого вкладеного списку.

Додавання, видалення та зміна загального списку, а також вкладених списків аналогічно до того, як це робиться зі звичайними (одномірними) списками:

```
people = [
    ["Tom", 29],
    ["Alice", 33],
    ["Bob", 27]
]
створення вкладеного списку
person = list()
person.append("Bill")
person.append(41)
# додавання вкладеного списку
people.append(person)
print(people[-1])# ["Bill", 41]
# додавання до вкладеного списку
people[-1].append("+79876543210")
print(people[-1])# ["Bill", 41, "+79876543210"]
видалення останнього елемента з вкладеного списку
people[-1].pop()
```

```
print(people[-1])# ["Bill", 41]
видалення всього останнього вкладеного списку
people.pop(-1)
# Зміна першого елемента
people[0] = ["Sam", 18]
print(people)# [ ["Sam", 18], ["Alice", 33], ["Bob", 27]]
```

Перебір вкладених списків:

```
people = [
    ["Tom", 29],
    ["Alice", 33],
    ["Bob", 27]
]
for person in people:
    for item in person:
        print(item, end=" | ")
```

Консольний висновок:

```
Том | 29 | Alice | 33 | Bob | 27 |
```

Списки та алгоритми

Списки нерідко використовуються для вирішення різних завдань, у яких, на перший погляд, у застосуванні списків немає сенсу, проте списки дозволяють спростити написання алгоритму або просто надати альтернативне рішення. Наприклад, візьмемо завдання щодо знаходження факторіалу числа:

```
fact = lambda n: [1,0] [n>1] або fact(n-1) * n
print(fact(4))
```

Для обчислення факторіалу тут визначено змінну fact, яка зберігає лямбда-вираз. Розглянемо його частинами. Спочатку йде визначення списку із двох елементів:

```
[1,0]
```

Далі йде звернення до елемента списку за допомогою індексу:

```
[1,0] [n>1]
```

Тут результат виразу `n>1` і представлятиме індекс у списку. Тобто якщо аргумент лямбда-виразу n більше 1, то вираз `n>1` дорівнює True або 1, а весь вираз `[1,0] [n>1]` поверне 0 (елемент за індексом 1). Якщо ж n дорівнює або менше 1, то вираз `n>1` дорівнює False або 0, а вираз `[1,0] [n>1]` поверне 1 (елемент за індексом 0)

Далі йде оператор `or`. Якщо вираз `[1,0] [n>1]` поверне 1 (якщо n<=1), то оператор `or` повертає це число 1. Якщо ж вираз `[1,0] [n>1]` поверне 0 (якщо n>1), то оператор `or` повертає результат виразу `fact(n-1) * n`, який рекурсивно викликає функцію fact, передаючи до неї число n-1. Через війну, якщо n=4, отримаємо таку послідовність действий:

- `[1,0] [4>1] or fact(4-1) * 4`
- `0 or fact(4-1) * 4`
- `([1,0] [3>1] or fact(3-1)) * 4`
- `(0 or fact(3-1)) * 4`
- `(([1,0] [2>1] or fact(2-1)) * 3) * 4`
- `((0 or fact(2-1)) * 3) * 4`
- `(([1,0] [1>1] or fact(1-1)) * 2) * 3) * 4`
- `(((1 or fact(1-1)) * 2) * 3) * 4`
- `(((1) * 2) * 3) * 4`

Додаткові матеріали

- Запитання для самоперевірки
- Вправи для самоперевірки

Кортежі
Кортеж (tuple) є послідовністю елементів, яка багато в чому схожа на список за тим винятком, що кортеж є незмінним (immutable) типом. Тому ми не можемо додавати або видаляти елементи кортежу, змінювати його.

Для створення кортежу використовуються круглі дужки, які містять його значення, розділені комами:

```
tom = ("Tom", 23)
print(tom)# ("Tom", 23)
```

Також для визначення кортежу ми можемо просто перерахувати значення через кому без застосування дужок:

```
tom = "Tom", 23
print(tom)# ("Tom", 23)
```

Якщо раптом кортеж складається з одного елемента, то після єдиного елемента кортежу необхідно поставити кому:

```
tom = ("Tom",)
```

Для створення кортежу з іншого набору елементів, наприклад зі списку, можна передати список у функцію tuple() , яка поверне кортеж:

```
data = ["Tom", 37, "Google"]
tom = tuple(data)
print(tom)# ("Tom", 37, "Google")
```

За допомогою вбудованої функції len() можна отримати довжину кортежу:

```
tom = ("Tom", 37, "Google")
print(len(tom))# 3
```

Звернення до елементів кортежу

Звернення до елементів у кортежі відбувається так само, як і в списку, за індексом. Індексація починається також з нуля при отриманні елементів з початку списку та з -1 при отриманні елементів з кінця списку:

```
tom = ("Tom", 37, "Google", "software developer")
print(tom[0])# Tom
print(tom[1])# 37
print(tom[-1])# software developer
```

Але оскільки кортеж - незмінний тип (immutable), ми зможемо змінити його елементи. Тобто наступний запис не працюватиме:

```
tom[1] = "Tim"
```

За потреби ми можемо розкласти кортеж на окремі змінні:

```
name, age, company, position = ("Tom", 37, "Google", "software
developer")
print(name)# Tom
print(age)# 37
print(position)# software developer
print(company)# Google
```

Отримання підкортежів

Як і в списках, можна отримати частину кортежу у вигляді іншого кортежу

```
tom = ("Tom", 37, "Google", "software developer")
# отримаємо підкортеж з 1 по 3 елементи (не включаючи)
print(tom[1:3])# (37, "Google")
# отримаємо підкортеж з 0 по 3 елементи (не включаючи)
print(tom[:3])# ("Tom", 37, "Google")
# отримаємо підкортеж з 1 до останнього елемент
print(tom[1:])# (37, "Google", "software developer")
```

Кортеж як параметр та результат функцій

Особливо зручно використовувати кортежі, коли необхідно повернути з функції одразу кілька значень. Коли функція повертає кілька значень, фактично вона повертає кортеж:

```
def get_user():
    name = "Tom"
    age = 22
    company = "Google"
    return name, age, company

user = get_user()
print(user)# ("Tom", 37, "Google")
```

Під час передачі кортежу в функцію за допомогою оператора * його можна розкласти на окремі значення, які передаються параметрам функції:

```
def print_person(name, age, company):
    print(f"Name: {name} Age: {age} Company: {company}")
tom = ("Tom", 22)
print_person(*tom, "Microsoft")# Name: Tom Age: 22 Company: Microsoft
bob = ("Bob", 41, "Apple")
print_person(*bob)# Name: Bob Age: 41 Company: Apple
```

Перебір кортежів

Для перебору кортежу можна використовувати стандартні цикли for та while . За допомогою циклу for:

```
tom = ("Tom", 22, "Google")
for item in tom:
```

```
    print(item)
```

За допомогою циклу while:

```
tom = ("Tom", 22, "Google")
i = 0
while i < len(tom):
    print(tom[i])
    i += 1
```

Перевірка наявності значення

Як для списку за допомогою виразу `элемент in кортеж` можна перевірити наявність елемента в кортежі:

```
user = ("Tom", 22, "Google")
name = "Tom"
if name in user:
    print("Користувача звуть Tom")
else:
    print("Користувач має інше ім'я")
```

Діапазони
Діапазони або range являють собою незмінний послідовний набір чисел. Для створення діапазів застосовується range , яка має такі форми:

- range(stop) : повертає всі цілі числа від 0 до stop
- range(start, stop) : повертає всі цілі числа в проміжку від start (включаючи) до stop (не включаючи).
- range(start, stop, step) : повертає цілі числа в проміжку від start (включаючи) до stop (не включаючи), які збільшуються на значення step

Приклади дзвінків функції range:

```
range(5)# 0, 1, 2, 3, 4
range(1, 5)# 1, 2, 3, 4
range(2, 10, 2)# 2, 4, 6, 8
range(10, 2, -2)# 10 8 6 4
```

Діапазони найчастіше застосовують у циклах for . Наприклад, виведемо послідовно всі числа від 0 до 4:

```
for i in range(5):
    print(i, end=" ")
```

```
# Консольний висновок
# 0, 1, 2, 3, 4
```

Інший приклад виведемо таблицю множення:

```
for i in range(1, 10):
    for j in range(1, 10):
        print(i * j, end="\t")
    print("\n")
```

```
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

Якщо нам необхідний послідовний список чисел, то для створення зручно використовувати функцію range :

```
numbers = list(range(10))
print(numbers)# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
numbers = list(range(2, 10))
print(numbers)# [2, 3, 4, 5, 6, 7, 8, 9]
numbers = list(range(10, 2, -2))
print(numbers)# [10, 8, 6, 4]
```

Перевагою діапазонів над стандартними списками і кортежами є те, що діапазон завжди займатиме одну і ту ж невелику кількість пам'яті незалежно від того, який набір чисел представляє цей діапазон. Насправді діапазон зберігає лише початкове, кінцеве значення та збільшення.

Додаткові матеріали

- Запитання для самоперевірки

Словники
Словник (dictionary) у мові Python зберігає колекцію елементів, де кожен елемент має унікальний ключ та асоційоване з ним певне значення.

Визначення словника має наступний синтаксис:

```
dictionary = { ключ 1: значення 1, ключ 2: значення 2, .... }
```

У фігурних дужках через кому визначається послідовність елементів, де для кожного елемента спочатку вказується ключ і через двокрапку його значення.

Визначимо словник:

```
users = {1: "Tom", 2: "Bob", 3: "Bill"}
```

У словнику users як ключі використовуються числа, а в якості значень - рядки. Тобто елемент із ключем 1 має значення "Tom", елемент із ключем 2 - значення "Bob" і т.д.

Інший приклад:

```
emails = {"tom@gmail.com": "Tom", "bob@gmai.com": "Bob",
"sam@gmail.com": "Sam"}
```

У словнику emails як ключі використовуються рядки - електронні адреси користувачів і як значення також рядки - імена користувачів.

Але необов'язково ключі та рядки мають бути однотипними. Вони можуть представляти різні типи:

```
objects = {1: "Tom", "2": True, 3: 100.6}
```

Ми можемо взагалі взагалі визначити порожній словник без елементів:

```
objects = {}
```

або так:

```
objects = dict()
```

Перетворення списків та кортежів на словник

Незважаючи на те, що словник і список - несхожі за структурою типи, проте існує можливості для окремих видів списків перетворення їх у словник за допомогою вбудованої функції dict() . Для цього список повинен зберігати набір вкладених списків. Кожен вкладений список повинен складатися з двох елементів – при конвертації у словник перший елемент стане ключем, а другий – значенням:

```
users_list = [
    ["+111123455", "Tom"],
    ["+384767557", "Bob"],
    ["+958758767", "Alice"]
]
users_dict = dict(users_list)
print(users_dict)# {"+111123455": "Tom", "+384767557": "Bob",
"+958758767": "Alice"}
```

Подібним чином можна перетворити на словник двомірні кортежі, які у свою чергу містять кортежі з двох елементів:

```
users_tuple = (
    ("+111123455", "Tom"),
    ("+384767557", "Bob"),
    ("+958758767", "Alice")
)
users_dict = dict(users_tuple)
print(users_dict)
```

Отримання та зміна елементів

Для звернення до елементів словника після його назви у квадратних дужках вказується ключ елемента:

```
dictionary[ключ]
```

Наприклад, отримаємо та змінимо елементи у словнику:

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
# отримуємо елемент із ключем "+11111111"
print(users["+11111111"])# Tom
# встановлення значення елемента з ключем "+33333333"
users["+33333333"] = "Bob Smith"
print(users["+33333333"])# Bob Smith
```

Якщо під час встановлення значення елемента з таким ключем у словнику не виявиться, то відбудеться його додавання:

```
users["+4444444"] = "Sam"
```

Але якщо ми спробуємо отримати значення з ключем, якого немає в словнику, Python згенерує помилку KeyError:

```
user = users["+4444444"]# KeyError
```

І щоб попередити цю ситуацію перед зверненням до елемента ми можемо перевіряти наявність ключа у словнику за допомогою виразу ключ in словник . Якщо ключ є у словнику, то цей вираз повертає True:

```
key = "+4444444"
if key in users:
    user = users[key]
    print(user)
else:
    print("Елемент не знайдений")
```

Також для отримання елементів можна використовувати метод get , який має дві форми:

- `get(key)` : повертає із словника елемент із ключем key. Якщо елемент з таким ключем немає, то повертає значення None
- `get(key, default)` : повертає із словника елемент із ключем key. Якщо елемент з таким ключем немає, то повертає значення за замовчуванням default

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
user1 = users.get("+55555555")
print(user1)# Alice
user2 = users.get("+33333333", "Unknown user")
print(user2)# Bob
user3 = users.get("+4444444", "Unknown user")
print(user3)# Unknown user
```

Вилучення

Для видалення елемента по ключу застосовується оператор del :

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
```

```
    "+55555555": "Alice"
}
del users["+55555555"]
print(users)# { "+11111111": "Tom", "+33333333": "Bob"}
```

Але варто враховувати, що якщо такого ключа не виявиться у словнику, то буде викинуто виняток KeyError. Тому знову ж таки перед видаленням бажано перевіряти наявність елемента з цим ключем.

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
key = "+55555555"
if key in users:
    del users[key]
    print(f"Елемент з ключем {key} видалений")
else:
    print("Елемент не знайдений")
```

Інший спосіб видалення представляє метод pop() . Він має дві форми:

- `pop(key)` : видаляє елемент за ключом key та повертає віддалений елемент. Якщо елемент із цим ключем відсутній, то генерується виняток KeyError
- `pop(key, default)` : видаляє елемент за ключом key та повертає віддалений елемент. Якщо елемент із цим ключем відсутній, то повертається значення default

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
key = "+55555555"
user = users.pop(key)
print(user)# Alice
user = users.pop("+44444444", "Unknown user")
print(user)# Unknown user
```

Якщо потрібно видалити всі елементи, то в цьому випадку можна скористатися методом clear():

```
users.clear()
```

Копіювання та об'єднання словників

Метод copy() копіює вміст словника, повертаючи новий словник:

```
users = {"+1111111": "Tom", "+3333333": "Bob", "+5555555": "Alice"}
students = users.copy()
print(students)# {"+1111111": "Tom", "+3333333": "Bob", "+5555555":
"Alice"}
```

Метод update() поєднує два словники:

```
users = {"+1111111": "Tom", "+3333333": "Bob"}
users2 = {"+2222222": "Sam", "+6666666": "Kate"}
users.update(users2)
print(users)# {"+1111111": "Tom", "+3333333": "Bob", "+2222222":
"Sam", "+6666666": "Kate"}
print(users2)# {"+2222222": "Sam", "+6666666": "Kate"}
```

При цьому словник users2 залишається без змін. Змінюється лише словник users, до якого додаються елементи іншого словника. Але якщо необхідно, щоб обидва вихідні словники були без змін, а результатом об'єднання був якийсь третій словник, можна попередньо скопіювати один словник в інший:

```
users3 = users.copy()
users3.update(users2)
```

Перебір словника

Для перебору словника можна скористатися циклом for:

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
for key in users:
    print(f"Phone: {key} User: {users[key]} ")
```

При переборі елементів ми отримуємо ключ поточного елемента і можемо отримати сам елемент.

Інший спосіб перебору елементів є використання методу items() :

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
for key, value in users.items():
    print(f"Phone: {key} User: {value} ")
```

Метод `items()` повертає набір кортежів. Кожен кортеж містить ключ і значення елемента, які при переборі ми можемо отримати в змінні key і value.

Також існують окремо можливості перебору ключів та перебору значень. Для перебору ключів ми можемо викликати у словника метод keys() :

```
for key in users.keys():
    print(key)
```

Щоправда, цей спосіб перебору немає сенсу, оскільки без виклику методу keys() ми можемо перебрати ключі, як було показано вище.

Для перебору значень ми можемо викликати у словника метод values() :

```
for value in users.values():
    print(value)
```

Комплексні словники

Крім найпростіших об'єктів типу чисел і рядків, словники також можуть зберігати і складніші об'єкти - ті ж списки, кортежі або інші словники:

```
users = {
    "Tom": {
        "phone": "+9714787745",
        "email": "tom12@gmail.com"
    },
    "Bob": {
        "phone": "+8763904444",
        "email": "bob@gmail.com",
        "skype": "bob123"
    }
}
```

У разі значення кожного елемента словника своєю чергою представляє окремий словник.

Для звернення до елементів вкладеного словника відповідно необхідно використовувати два ключі:

```
old_email = users["Tom"]["email"]
users["Tom"]["email"] = "supertom@gmail.com"
print(users["Tom"])# { phone": "+9714787745", "email":
"supertom@gmail.com }
```

Але якщо ми спробуємо отримати значення за ключом, який відсутній у словнику, Python згенерує виняток KeyError:

```
tom_skype = users["Tom"]["skype"]# KeyError
```

Щоб уникнути помилки, можна перевіряти наявність ключа у словнику:

```
key = "skype"
if key in users["Tom"]:
    print(users["Tom"]["skype"])
else:
    print("skype is not found")
```

Або як альтернативу можна використовувати метод get() :

```
users = {
    "Tom": {
        "phone": "+9714787745",
        "email": "tom12@gmail.com"
    }
}
tom_skype = users["Tom"].get("skype", "Undefined")
print(tom_skype)# Undefined
```

Додаткові матеріали

- Запитання для самоперевірки
- Вправи для самоперевірки

Безліч
Безліч (set) є ще одним видом набору, який зберігає тільки унікальні елементи. Для визначення множини використовуються фігурні дужки, в яких перераховуються елементи:

```
users = {"Tom", "Bob", "Alice", "Tom"}
print(users)# {"Alice", "Bob", "Tom"}
```

Зверніть увагу, що незважаючи на те, що функція print вивела один раз елемент "Tom", хоча у визначенні множини цей елемент міститься двічі. Все тому, що безліч містить тільки унікальні значення.

Також для визначення множини може застосовуватися функція set() , до якої передається список або кортеж елементів:

```
people = ["Mike", "Bill", "Ted"]
users = set(people)
print(users)# {"Mike", "Bill", "Ted"}
```

Функцію set зручно застосовувати для створення порожньої множини:

```
users = set()
```

Для отримання довжини множини застосовується вбудована функція len() :

```
users = {"Tom", "Bob", "Alice"}
print(len(users))# 3
```

Додавання елементів

Для додавання одиночного елемента викликається метод add() :

```
users = set()
users.add("Sam")
print(users)
```

Видалення елементів

Для видалення одного елемента викликається метод remove() , який передається видалений елемент. Але слід враховувати, що якщо такого елемента не виявиться у множині, то буде згенерована помилка. Тому перед видаленням слід перевіряти наявність елемента за допомогою оператора in :

```
users = {"Tom", "Bob", "Alice"}
user = "Tom"
if user in users:
    users.remove(user)
print(users)# {"Bob", "Alice"}
```

Також для видалення можна використовувати метод discard() , який не генеруватиме виключення за відсутності елемента:

```
users = {"Tom", "Bob", "Alice"}
users.discard("Tim")# елемент "Tim" відсутній, і метод нічого не робить
print(users)# {"Tom", "Bob", "Alice"}
users.discard("Tom")# елемент "Tom" є, і метод видаляє елемент
print(users)# {"Bob", "Alice"}
```

Для видалення всіх елементів викликається метод clear() :

```
users.clear()
```

Перебір множини

Для перебору елементів можна використовувати цикл:

```
users = {"Tom", "Bob", "Alice"}
for user in users:
    print(user)
```

При переборі кожен елемент міститься у змінну user.

Операції з безліччю

За допомогою методу copy() можна скопіювати вміст однієї множини в іншу змінну:

```
users = {"Tom", "Bob", "Alice"}
students = users.copy()
print(students)# {"Tom", "Bob", "Alice"}
```

Об'єднання множин

Метод union() поєднує дві множини і повертає нову множину:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
users3 = users.union(users2)
print(users3)# {"Bob", "Alice", "Sam", "Kate", "Tom"}
```

Замість методу `union()` ми могли використовувати операцію логічного складання - | :

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
print(users | users2)# {"Bob", "Alice", "Sam", "Kate", "Tom"}
```

Перетин множин

Перетин множин дозволяє отримати тільки ті елементи, які є одночасно в обох множинах. Метод intersection() здійснює операцію перетину множин і повертає нову множину:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
users3 = users.intersection(users2)
print(users3)# {"Bob"}
```

Замість методу intersection ми могли б використовувати операцію логічного множення:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
print(users & users2)# {"Bob"}
```

У цьому випадку ми отримали б той самий результат.

Модифікація методу - intersection_update() замінює пересіченими елементами перше безліч:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
users.intersection_update(users2)
print(users)# {"Bob"}
```

Різниця множин

Ще одна операція – різниця множин повертає ті елементи, які є в першій множині, але відсутні в другій. Для отримання різниці множин можна використовувати метод difference або операцію віднімання:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
```

```
users3 = users.difference(users2)
print(users3)# {"Tom", "Alice"}
print(users - users2)# {"Tom", "Alice"}
```

Окремий різновид різниці множин - симетрична різниця проводиться за допомогою методу symmetric_difference() або за допомогою операції `^`. Вона повертає всі елементи обох множин за винятком загальних:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
users3 = users.symmetric_difference(users2)
print(users3)# {"Tom", "Alice", "Sam", "Kate"}
users4 = users ^ users2
print(users4)# {"Tom", "Alice", "Sam", "Kate"}
```

Відносини між множинами

Метод issubset дозволяє з'ясувати, чи є поточна множина підмножиною (тобто частиною) іншої множини:

```
users = {"Tom", "Bob", "Alice"}
superusers = {"Sam", "Tom", "Bob", "Alice", "Greg"}
print(users.issubset(superusers))# True
print(superusers.issubset(users))# False
```

Метод issuperset , навпаки, повертає True, якщо поточна множина є надмножиною (тобто містить) для іншої множини:

```
users = {"Tom", "Bob", "Alice"}
superusers = {"Sam", "Tom", "Bob", "Alice", "Greg"}
print(users.issuperset(superusers))# False
print(superusers.issuperset(users))# True
```

frozen set

Тип frozen set є видом множин, які не можуть бути змінені. Для його створення використовується функція
frozenset :

```
users = frozenset({"Tom", "Bob", "Alice"})
```

У функцію frozenset передається набір елементів - список, кортеж, інше безліч.

У таку множину ми не можемо додати нові елементи, як і видалити з нього вже наявні. Саме тому frozen set підтримує обмежений набір операцій:

- `len(s)` : повертає довжину множини
- `x in s` : повертає True, якщо елемент x присутній у множині s
- `x not in s` : повертає True, якщо елемент x відсутній у безлічі s
- `s.issubset(t)` : повертає True, якщо t містить безліч s
- `s.issuperset(t)` : повертає True, якщо t міститься у множині s
- `s.union(t)`

  : повертає об'єднання множин s і t
- `s.intersection(t)` : повертає перетин множин s і t
- `s.difference(t)` : повертає різницю множин s і t
- `s.copy()` : повертає копію множини s

Додаткові матеріали

- Запитання для самоперевірки
- Вправи для самоперевірки

List comprehension
Функціональність list comprehension надає більш короткий та лаконічний синтаксис для створення списків на основі інших наборів даних. Вона має наступний синтаксис:

```
newlist = [expression for item in iterable (if condition)]
```

Синтаксис `list comprehension` складається з наступних компонентів:

- `iterable` : джерело даних, що перебирається, в якості якого може виступати список, безліч, послідовність, або навіть функція, яка повертає набір даних, наприклад, `range()`
- `item` : елемент, що витягується з джерела даних
- `expression` : вираз, який повертає певне значення Це значення потім потрапляє в список, що генерується
- `condition` : умова, якій повинні відповідати елементи, що витягуються з джерела даних. Якщо елемент НЕ задовольняє умову, він не вибирається. Необов'язковий параметр.

Розглянемо невеликий приклад. Припустимо, нам треба вибрати зі списку всі числа, які більше 0. У випадку ми могли б написати так:

```
numbers = [-3, -2, -1, 0, 1, 2, 3]
positive_numbers = []
```

```
for n in numbers:
    if n > 0:
        positive_numbers.append(n)
print(positive_numbers)# [1, 2, 3]
```

Тепер змінимо цей код, застосувавши `list comprehension`:

```
numbers = [-3, -2, -1, 0, 1, 2, 3]
positive_numbers = [n for n in numbers if n > 0]

print(positive_numbers)# [1, 2, 3]
```

Вираз `[n for n in numbers if n > 0]` каже вибрати зі списку numbers кожен елемент змінну n, якщо n більше 0 і повернути n в результуючий список.

джерело даних iterable

Як джерело даних `iterable` може використовуватися будь-який об'єкт, що перебирається, наприклад, інший список, словник і т.д. Наприклад, функція `range()`
повертає все числячи нуля до зазначеного порога не включаючи:

```
numbers = [n for n in range(10)]
print(numbers)# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Нерідко ця конструкція застосовується, щоб створити зі словника список. Наприклад, виберемо зі словника всі ключі:

```
dictionary = {"red": "червоний", "blue": "синій", "green": "зелений"}
words = [word for word in dictionary]
print(words)# ['red', 'blue', 'green']
```

Повернення результату

Параметр `expression` являє собою вираз, який повертає деяке значення. Це значення потім поміщається в список, що генерується. У прикладах вище це був поточний елемент, який витягується із джерела даних:

```
numbers = [-3, -2, -1, 0, 1, 2, 3]
new_numbers = [n for n in numbers]
print(new_numbers)# [-3, -2, -1, 0, 1, 2, 3]
```

Так, в даному випадку параметр `expression` представляє елемент, що безпосередньо витягується зі списку numbers n. Але це можуть бути і складніші значення. Наприклад, повернемо подвоєне значення числа:

```
numbers = [-3, -2, -1, 0, 1, 2, 3]
new_numbers = [n * 2 for n in numbers]
print(new_numbers)# [-6, -4, -2, 0, 2, 4, 6]
```

Тут `expression` репрезентує вираз `n * 2`

Це можуть бути і складніші вирази:

```
numbers = [-3, -2, -1, 0, 1, 2, 3]
new_numbers = [n * 2 if n > 0 else n for n in numbers]
print(new_numbers)# [-3, -2, -1, 0, 2, 4, 6]
```

Тут параметр `expression` представляє вираз `n * 2 if n > 0 else n`. У разі ми говоримо повернути значення n * 2, якщо n > 0, інакше повернути n.

Можна проводити різні трансформації `expression` з даними. Наприклад, повернемо також із словника значення за ключом:

```
dictionary = {"red": "червоний", "blue": "синій", "green": "зелений"}
words = [f"{key}: {dictionary[key]}" for key in dictionary]
print(words)# ['red: червоний', 'blue: синій', 'green: зелений']
```

Умова

Умова - параметр `condition` визначає фільтр для вибору елементів із джерела даних. Застосуємо умову для конкретизації вибірки, наприклад, виберемо лише парні числа:

```
numbers = [n for n in range(10) if n % 2 == 0]
print(numbers)# [0, 2, 4, 6, 8]
```

Виберемо тільки ключі зі словника, довжина яких більше 3:

```
dictionary = {"red": "червоний", "blue": "синій", "green": "зелений"}
words = [f"{key}: {dictionary[key]}" for key in dictionary if
len(key) > 3]
print(words)# ['blue: синій', 'green: зелений']
```

Додаткові матеріали

- Запитання для самоперевірки

Упаковка та розпакування

Розпакування

Розпакування (unpacking , також звана Деструктуризація) представляє розкладання колекції (кортежу, списку і т.д.) на окремі значення.

Так само, як і багато мов програмування, Python підтримує концепцію множинного присвоєння. Наприклад:

```
x, y = 1, 2
print(x)# 1
print(y)# 2
```

В даному випадку надамо значення відразу двом змінним. Присвоєння йде за позицією: змінна x отримує значення 1, а змінна y - значення 2.

Даний приклад насправді вже представляє деструктуризацію чи розпакування. Значення `1, 2` фактично є кортежем, оскільки саме коми між значеннями свідчать, що це кортеж. І ми також могли б написати так:

```
x, y = (1, 2)
print(x)# 1
print(y)# 2
```

У будь-якому випадку ми маємо справу з деструктуризацією, коли перший елемент кортежу передається першою змінною, другий елемент - другою змінною і таке інше. Тобто розкладання йде за позицією.

Подібним чином можна розкласти інші кортежі, наприклад:

```
name, age, company = ("Tom", 38, "Google")
print(name)# Tom
print(age)# 38
print(company)# Google
```

Тільки кортежами ми не обмежені і можемо "розпаковувати" та інші колекції, наприклад, списки:

```
people = ["Tom", "Bob", "Sam"]
first, second, third = people
print(first)# Tom
print(second)# Bob
print(third)# Sam
```

При розкладанні словника змінні одержують ключі словника:

```
dictionary = {"red": "червоний", "blue": "синій", "green": "зелений"}
r, b, g = dictionary
print(r)# red
print(b)# blue
print(g)# green
# отримуємо значення за ключем
print(dictionary[g])# зелений
```

Деструктуризація у циклах

Цикли Python дозволяють розкласти колекції на окремі складові:

```
people = [
    ("Tom", 38, "Google"),
    ("Bob", 42, "Microsoft"),
    ("Sam", 29, "JetBrains")
]
for name, age, company in people:
    print(f"Name: {name}, Age: {age}, Company: {company}")
```

Тут ми перебираємо список кортежів людей. Кожен кортеж складається з трьох елементів, відповідно при переборі ми можемо передати їх у змінні name, age і company.

Інший приклад - функція enumerate() . Вона приймає як параметр колекцію, створює для кожного елемента кортеж і повертає набір з подібних кортежів. Кожен кортеж містить індекс, який збільшується з кожною ітерацією:

```
people = ["Tom", "Bob", "Sam"]
для index, name in enumerate(people):
    print(f"{index}.{name}")
# результат
# 0.Tom
# 1.Bob
# 2.Sam
```

Ігнорування значень

Якщо якийсь елемент колекції не потрібен, то зазвичай йому визначається змінна з ім'ям _ (прочерк):

```
person =("Tom", 38, "Google")
name, _, company = person
print(name)# Tom
print(company)# Google
```

Тут нам важливий другий елемент кортежу, тому йому визначаємо змінну _. Хоча насправді _ - таке ж дійсне ім'я, як name і company:

```
name, _, company = person
print(_)# 38
```

Упаковка значень та оператор *

Оператор упаковує значення в колекцію. Наприклад:

```
num1=1
num2=2
num3=3
*numbers,=num1,num2,num3
print(numbers)#[1, 2, 3]
```

Тут ми упаковуємо значення з кортежу (num1,num2,num3) до списку numbers. Причому, щоб отримати список, після numbers вказується кома.

Як правило, упаковка застосовується для збирання значень, що залишилися після присвоєння результатів деструктуризації. Наприклад:

```
head, *tail = [1, 2, 3, 4, 5]
print(head)# 1
print(tail)# [2, 3, 4, 5]
```

Тут змінна `head` відповідно до позиції отримує перший елемент списку. Всі інші елементи передаються в змінну `tail`. Таким чином, змінна `tail` буде представляти список з елементів, що залишилися.

Аналогічним чином можна отримати все, крім останнього:

```
*head, tail = [1, 2, 3, 4, 5]
print(head)# [1, 2, 3, 4]
print(tail)# 5
```

Або елементи посередині, крім першого і останнього:

```
head, * middle, tail = [1, 2, 3, 4, 5]
print(head)# 1
print(middle)# [2, 3, 4]
print(tail)# 5
```

Або всі крім першого та другого:

```
first, second, *other = [1, 2, 3, 4, 5]
print(first)# 1
print(second)# 2
print(other)# [3, 4, 5]
```

Втім, таким чином ми можемо отримувати різні комбінації елементів колекції. Причому не лише списків, а й кортежів, словників та інших колекцій.

Інший приклад - нам треба отримати лише перший, третій та останній елемент, а решта елементів нам не потрібна. Загалом ми повинні надати змінні для всіх елементів колекції. Однак якщо колекція має 100 елементів, а нам потрібно лише три, не будемо ми визначати всі сто змінних. І в цьому випадку знову ж таки можна застосувати упаковку:

```
first, _, third, *_, last = [1, 2, 3, 4, 5, 6, 7, 8]
print(first)# 1
print(third)# 3
print(last)# 8
```

Також можна отримати ключі словника:

```
red, *other, green = {"red":"червоний", "blue":"синій",
"yellow":"жовтий", "green":"зелений"}
print(red)# red
print(green)# green
print(other)# ['blue', 'yellow']
```

Розпакування та оператори * та **

Оператор * разом із оператором ** також може застосовуватися для розпакування значень. Оператор * використовується для розпакування кортежів, списків, рядків, множин, а оператор ** - для розпакування словників. Особливо це може бути корисним, коли на основі одних колекцій створюються інші. Наприклад, розпакування кортежів та списків:

```
nums1 = [1, 2, 3]
nums2 = (4, 5, 6)
# Розпаковуємо список nums1 і кортеж nums2
nums3 = [*nums1, *nums2]
print(nums3)# [1, 2, 3, 4, 5, 6]
```

Тут розпаковуємо значення зі списку nums1 та кортежу nums2 та поміщаємо їх у список nums3.

Подібно розкладаються словники, тільки застосовується оператор ** :

```
dictionary1 = {"red":"червоний", "blue":"синій"}
dictionary2 = {"green":"зелений", "yellow":"жовтий"}
# Розпаковуємо словники
dictionary3 = {**dictionary1, **dictionary2}
print(dictionary3)# {'red': 'червоний', 'blue': 'синій', 'green':
'зелений', 'yellow': 'жовтий'}
```

Упаковка та розпакування у параметрах функцій
Однією з найпоширеніших сфер, де застосовуються упаковка та розпакування – це параметри функцій. Так, у визначеннях різних функцій нерідко можна побачити, що вони набувають таких параметрів як `*args` і `**kwargs` .

Терміни args і kwargs — це угоди щодо програмування на Python, насправді замість них можна використовувати будь-які іменування.
*args представляє параметри, що передаються за позицією. А **kwargs означає параметри, що передаються на ім'я. означає аргументи ключового слова.

Оператор * застосовується з будь-яким об'єктом, що ітерується (наприклад, кортежем, списком і рядками). Тоді як оператор **
можна використовувати лише зі словниками.

*args

Оператор дозволяє передати в функцію кілька значень, і всі вони будуть упаковані в кортеж:

```
def fun(*args):
   # звертаємось до першого елементу кортежу
    print(args[0])

   # виводимо весь кортеж
    print(args)
```

```
fun("Python", "C++", "Java", "C#")
```

Тут функція `fun` набуває кортежу значень. При виклик ми можемо передати їй різну кількість значень. Так, у прикладі вище передається чотири рядки, які утворюють кортеж. Консольний висновок програми:

```
python
('Python', 'C++', 'Java', 'C#')
```

Завдяки такій можливості ми можемо передавати у функцію змінну кількість значень:

```
def sum(*args):
    result = 0
    for arg in args:
        result += arg
    return result

print(sum(1, 2, 3))# 6
print(sum(1, 2, 3, 4))# 10
print(sum(1, 2, 3, 4, 5))# 15
```

Оператор**

Оператор ** упаковує аргументи, передані на ім'я, у словник. Імена параметрів є ключами. Наприклад, визначимо функцію, яка просто виводитиме всі передані параметри

```
def fun(**kwargs):
    print(kwargs)# виводимо словник на консоль

fun(name="Tom", age="38", company="Google")
fun(language="Python", version="3.11")
```

Консольний висновок програми:

```
{'name': 'Tom', 'age': '38', 'company': 'Google'}
{'language': 'Python', 'version': '3.11'}
```

Оскільки аргументи передаються у функцію як словника, то всередині функції через ключі ми можемо отримати їх значення:

```
def fun(**kwargs):
    for key in kwargs:
        print(f"{key} = {kwargs[key]}")

fun(name="Tom", age="38", company="Google")
```

Консольний висновок програми:

```
name = Tom
age = 38
company = Google
```

Розпакування аргументів

Вище було описано, як оператори * та ** застосовуються для пакування аргументів у кортеж та словник відповідно. Але ці оператори можуть використовуватися для розпакування.

Оператор * та розпакування

Спочатку розглянемо ситуацію, де це може стати в нагоді. Нехай ми передаємо в функцію кортеж:

```
def sum(*args):
  result = 0
  for arg in args:
    result += arg
  return result

numbers = (1, 2, 3, 4, 5)
print(sum(numbers))
```

Тут у виклик функції sum передається кортеж. Параметр `*args` по суті також представляє кортеж, і здається, все має працювати. Тим не менш, ми зіткнемося з помилкою

```
TypeError: unsupported operand type(s) for +=: 'int' and 'tuple'
```

Тобто в даному випадку кортеж numbers передається як елемент кортежу `*args`.

І щоб елементи кортежу були передані в кортеж `*args` як окремі значення, необхідно виконати їхню розпакування:

```
def sum(*args):
  result = 0
  for arg in args:
    result += arg
  return result

numbers = (1, 2, 3, 4, 5)
# застосовуємо розпакування - *numbers
print(sum(*numbers))# 15
```

Тут при передачі кортежу numbers у функцію sym застосовується розпакування: `*numbers`

Іншим випадком розпакування може бути ситуація, коли функція приймає кілька параметрів, а ми передаємо один кортеж або список:

```
def print_person(name, age, company):
  print(f"Name:{name}, Age: {age}, Company: {company}")

person =("Tom", 38, "Google")
# виконуємо розпаковування кортежу person
print_person(*person)# Name:Tom, Age: 38, Company: Google
```

В даному випадку вираз `*person` розкладає кортеж person на окремі значення, що передаються параметрам name, age та company.

Оператор ** та розпакування

Оператор ** застосовується для розпакування словників:

```
def print_person(name, age, company):
  print(f"Name:{name}, Age: {age}, Company: {company}")

tom ={"name":"Tom", "age":38, "company":"Google"}
# виконуємо розпакування словника tom
print_person(**tom)# Name:Tom, Age: 38, Company: Google
```

Тут вираз `**tom` розкладає словник окремі значення, які передаються параметрам name, age і company за назвою ключів.

Поєднання параметрів

Параметри `*args` можуть `*kwargs` використовуватися в функції разом з іншими параметрами. Наприклад:

```python
def sum(num1, num2, *nums):
    result=num1+num2
    for n in nums:
        result += n
    return result
print(sum(1,2,3))# 6
print(sum(1,2,3,4))# 10
```

Розділ 6. Модулі

Визначення та підключення модулів

Модуль у мові Python представляє окремий файл із кодом, який можна повторно використовувати в інших програмах.

Для створення модуля необхідно створити власне файл з розширенням *.py , який представлятиме модуль. Назва файлу представлятиме назву модуля. Потім у цьому файлі слід визначити одну або кілька функцій.

Допустимо, основний файл програми називається main.py . І хочемо підключити до нього зовнішні модулі.

Для цього спочатку визначимо новий модуль: створимо в тій же папці, де знаходиться main.py новий файл, який назвемо message.py . За умовчанням інтерпретатор Python шукає модулі за рядом стандартних шляхів, один з яких - це папка головного скрипта, що запускається. Тому щоб інтерпретатор підхопив модуль message.py, для простоти обидва файли помістимо в один проект.

 Модулі в Python

Відповідно модуль буде називатися message . Визначимо в ньому наступний код:

```python
hello = "Hello all"

def print_message(text):
    print(f"Message: {text}")
```

Тут визначено змінну hello та функцію print_message, яка як параметр отримує деякий текст і виводить його на консоль.

В основному файлі програми - main.py використовуємо цей модуль:

```python
import message# підключаємо модуль message
# виводимо значення змінної hello
print(message.hello)# Hello all
# звертаємось до функції print_message
message.print_message("Hello work")# Message: Hello work
```

Для використання модуля його слід імпортувати за допомогою оператора import , після якого вказується ім'я модуля: `import message` .

Щоб звертатися до функціональності модуля, нам потрібно отримати його простір імен . За умовчанням воно збігатиметься з ім'ям модуля, тобто в нашому випадку також називатиметься message .

Отримавши простір імен модуля, ми зможемо звернутися до його функцій за схемою

простір_імен.функція

Наприклад, звернення до функції `print_message()` з модуля message:

```
message.print_message("Hello work")
```

І після цього ми можемо запустити головний скрипт main.py і він задіює модуль message.py. Зокрема, консольний висновок буде таким:

```
Hello all
Message: Hello work
```

Підключення функціональності модуля до глобального простору імен

Інший варіант налаштування передбачає імпорт функціональності модуля у глобальний простір імен поточного модуля за допомогою ключового слова from :

```
від message import print_message
# звертаємось до функції print_message з модуля message
print_message("Hello work")# Message: Hello work
# Змінна hello з модуля message не доступна, оскільки вона не
імпортована
# print(message.hello)
# print(hello)
```

У даному випадку ми імпортуємо з модуля message в глобальний простір імен функцію `print_message()` . Тому ми зможемо її використовувати без вказівки простору імен модуля ніби вона була визначена в цьому ж файлі.

Всі інші функції, змінні з модуля недоступні (наприклад, у прикладі вище змінна hello). Якщо ми хочемо їх також використовувати, їх можна підключити окремо:

```
від message import print_message
від message import hello
```

```
# звертаємось до функції print_message з модуля message
print_message("Hello work")# Message: Hello work
# звертаємось до змінної hello з модуля message
print(hello)# Hello all
```

Якщо необхідно імпортувати в глобальний простір імен весь функціонал, замість назв окремих функцій і змінних можна використовувати символ зводочки * :

```
від message import *
# звертаємось до функції print_message з модуля message
print_message("Hello work")# Message: Hello work
# звертаємось до змінної hello з модуля message
print(hello)# Hello all
```

Але варто зазначити, що імпорт у глобальний простір імен загрожує колізіями імен функцій. Наприклад, якщо в тому ж файлі визначена функція з тим самим ім'ям до її виклику, то буде викликатися функція, яка визначена останньою:

```
від message import *

print_message("Hello work")# Message: Hello work - застосовується
функція з модуля message
def print_message(some_text):
    print(f"Text: {some_text}")

print_message("Hello work")# Text: Hello work - застосовується
функція з поточного файлу
```

Таким чином, однойменна функція поточного файлу приховує функцію підключеного модуля.

Встановлення псевдонімів

При імпорті модуля та його функціональності ми можемо встановити їм псевдоніми. Для цього застосовується ключове слово as після якого вказується псевдонім. Наприклад, встановимо псевдонім для модуля:

```
import message as mes# модуль message проектується на псевдонім mes
# виводимо значення змінної hello
print(mes.hello)# Hello all
# звертаємось до функції print_message
mes.print_message("Hello work")# Message: Hello work
```

В даному випадку простір імен буде називатися mes , і через цей псевдонім можна звертатися до функціональності модуля.

Так само можна встановити псевдоніми для окремої функціональності модуля:

```
from message import print_message as display
from message import hello as welcome
print(welcome)# Hello all - змінна hello з модуля message
display("Hello work")# Message: Hello work - функція print_message з
модуля message
```

Тут для функції print_message з модуля message встановлюється псевдонім display, а змінної hello - псевдонім welcome. І через ці псевдоніми ми зможемо до них звертатись.

Псевдоніми можуть бути корисні, коли нас не влаштовують імена функцій та змінних, наприклад, вони надто довгі, і ми хочемо їх скоротити, або ми хочемо дати їм більш описові, на наш погляд, імена. Або якщо у поточному файлі вже є функціональність із тими самими іменами, і з допомогою встановлення псевдонімів ми можемо уникнути конфлікту імен. Наприклад:

```
from message import print_message as display
def print_message(some_text):
    print(f"Text: {some_text}")
# функція print_message із модуля message
display("Hello work")# Message: Hello work
# функція print_message із поточного файлу
print_message("Hello work")# Text: Hello work
```

Ім'я модуля

У прикладі вище модуль main.py, який є основним, використовує модуль message.py. При запуску модуля main.py програма виконає всю потрібну роботу. Однак якщо ми запустимо окремо модуль message.py сам по собі, то нічого на консолі не побачимо. Адже модуль message просто визначає функцію та змінну і не виконує жодних інших дій. Але ми можемо зробити так, щоб модуль message.py міг використовуватися як сам собою, так і підключатися в інші модулі.

При виконанні модуля середовище визначає його ім'я та надає його глобальній змінній __name__ (з обох сторін по два підкреслення). Якщо модуль запускається, то його ім'я дорівнює __main__ (також по два підкреслення з кожної сторони). Якщо модуль використовується в іншому модулі, то в момент виконання його ім'я аналогічне назві файлу без розширення py. І ми можемо це використати. Так, змінимо вміст файлу message.py :

```
hello = "Hello all"

def print_message(text):
    print(f"Message: {text}")

def main():
    print_message(hello)

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

У цьому випадку модуль message.py для тестування функціональності модуля додано функцію `main`. І ми можемо одразу запустити файл message.py окремо від усіх та протестувати код.

Слід звернути увагу на виклик функції main:

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

Змінна __name__ вказує на ім'я модуля. Для головного модуля, який безпосередньо запускається, ця змінна завжди матиме значення __main__ незалежно від імені файлу.

Тому, якщо ми будемо запускати скрипт message.py окремо сам по собі, то Python надасть змінній __name__ значення `__main__`, далі у виразі if викличе функцію main з цього ж файлу.

Однак якщо ми будемо запускати інший скрипт, а цей - message.py - будемо підключати як допоміжний, для message.py змінна __name__ матиме значення `message`. І відповідно метод main у файлі message.py не працюватиме.

Цей підхід з перевіркою імені модуля є рекомендованим підходом, ніж просто виклик методу main.

У файлі main.py також можна зробити перевірку на те, чи модуль головним (хоча в принципі це необов'язково):

```
import message

def main():
    message.print_message("Hello work")# Message: Hello work

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

Python надає низку вбудованих модулів, які ми можемо використовувати у своїх програмах. У наступних статтях розглянемо основні їх.

Генерація байткоду модулів
При виконанні скрипта мовою Python все виконання в загальному випадку розбивається на дві стадії:

1. Файл з кодом (файл з розширенням .py) компілюється в проміжний байткод.
2. Далі скомпільований байткод інтерпретується, тобто відбувається власне виконання програми

При цьому нам не треба явно генерувати ніякий байткод, він створюється неявно при виконанні скрипту Python. Якщо програма імпортує зовнішні модулі/бібліотеки і вони імпортуються вперше, їх скомпільований байткод зберігається зберігається у файлі з розширенням .pyc і кешується в каталозі __pycache__ в папці, де розташований файл з кодом python. Якщо ми вносимо у вихідний файл бібліотеки зміни, Python перекомпілює файл байткоду. Якщо змін у коді немає, то завантажується раніше скомпільований байткод із файлу *.pyc . Це дозволяє оптимізувати роботу з додатком, швидше його компілювати та виконувати.

Однак байткод основного скрипта, який представляє основний файл програми і який передається інтерпретатору python, не зберігається у файлі *.pyc і перекомпілюється щоразу під час запуску програми.

Допустимо, у папці проекту у нас розміщений файл user.py з найпростішою функцією, яка приймає два параметри і виводить їх значення:

```
def printUser(username, userage):
    print(f"Name: {username} Age:{userage}")
```

Підключимо цей файл у головному модулі програми, який нехай називається app.py :

```
import user
username = "Tom"
userage = 39
user.printUser(username, userage)
```

При виконанні цього скрипта в папці проекті (де розташовується модуль "user.py") буде створено каталог __pycache__ . А в ньому буде згенерований файл байткоду, який буде на кшталт наступного `user.cpython-версия.pyc` , де в якості версії буде застосовуватися версія інтерпретатора, що використовується, наприклад, `311` (для версії Python 3.11). Згенерований pyc -файл є бінарним, тому текстовому редакторі немає сенсу його відкривати.

 __pycache__ та модулі в мові програмування Python

Ручна компіляція байткоду

Хоча файл байткоду створюється автоматично, ми можемо вручну його згенерувати. Для цього є кілька способів: компіляція за допомогою скрипта py_compile та компіляція за допомогою модуля compileall .

Скрипт py_compile використовується для компіляції окремих файлів. Для компіляції довільного скрипта user.py у файл із байткодом ми могли б використовувати таку програму:

```
import py_compile
py_compile.compile("user.py")# передаємо шлях до скрипту
```

Для компіляції у функцію `compile()` передаємо шлях до скрипту. Після виконання програми в поточній папці також буде згенеровано каталог __pycache__ , а в ньому файл user.cpython-311.pyc

Модуль compileall застосовується для компіляції всіх файлів Python за певними шляхами. Наприклад, скомпілюємо всі файли в каталозі C:/python/files

```
python -m compileall c:\python\files
```

За замовчуванням компілюються навіть файли, які містяться в підкаталогах. Якщо потрібно скомпілювати тільки ті файли, які розташовуються безпосередньо у зазначеній папці, то застосовується опція -l

```
python -m compileall c:\python\files -l
```

Модуль random
Модуль random управляє генерацією випадкових чисел. Його основні функції:

- random() : генерує випадкове число від 0.0 до 1.0
- randint() : повертає випадкове число з певного діапазону
- randrange() : повертає випадкове число з певного набору чисел
- shuffle() : перемішує список
- choice() : повертає випадковий елемент списку

Функція random() повертає випадкове число з точкою, що плаває, в проміжку від 0.0 до 1.0. Якщо нам необхідно число з більшого діапазону, скажімо від 0 до 100, ми можемо відповідно помножити результат функції random на 100.

```
import random
number = random.random()# значення від 0.0 до 1.0
print(number)
number = random.random() * 100# значення від 0.0 до 100.0
print(number)
```

Функція randint(min, max) повертає ціле випадкове число в проміжку між двома значеннями min і max.

```
import random
number = random.randint(20, 35)# значення від 20 до 35
print(number)
```

Функція randrange() повертає ціле випадкове число з певного набору чисел. Вона має три форми:

- `randrange(stop)` : як набор чисел, з яких відбувається вилучення випадкового значення, буде використовуватися діапазон від 0 до числа stop
- `randrange(start, stop)` : набір чисел представляє діапазон від числа start до числа stop
- `randrange(start, stop, step)` : набір чисел представляє діапазон від числа start до числа stop, при цьому кожне число в діапазоні відрізняється від попереднього на крок step

```
import random
number = random.randrange(10)# значення від 0 до 10 не включаючи
print(number)
number = random.randrange(2, 10)# значення в діапазоні 2, 3, 4, 5, 6,
7, 8, 9
print(number)
number = random.randrange(2, 10, 2)# значення в діапазоні 2, 4, 6, 8
print(number)
```

Робота зі списком

Для роботи зі списками у модулі random визначено дві функції: функція shuffle() перемішує список випадковим чином, а функція choice() повертає один випадковий елемент зі списку:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
random.shuffle(numbers)
print(numbers)
```

```
random_number = random.choice(numbers)
print(random_number)
```

Математичні функції та модуль math

Вбудований модуль math у Python надає набір функцій для виконання математичних, тригонометричних та логарифмічних операцій. Деякі з основних функцій модуля:

- pow(num, power) : зведення числа num у ступінь power
- sqrt(num) : квадратний корінь числа num
- ceil(num) : округлення числа до найближчого найбільшого цілого
- floor(num) : округлення числа до найближчого найменшого цілого
- factorial(num) : факторіал числа
- degrees(rad) : переклад з радіан у градуси
- radians(grad) : переведення з градусів у радіани
- cos(rad) : косинус кута в радіанах
- sin(rad) : синус кута в радіанах
- tan(rad) : тангенс кута в радіанах
- acos(rad) : арккосинус кута в радіанах
- asin(rad) : арксинус кута в радіанах
- atan(rad) : арктангенс кута в радіанах
- log(n, base) : логарифм числа n на основі base
- log10(n) : десятковий логарифм числа n

Приклад застосування деяких функцій:

```
import math
# Зведення числа 2 у ступінь 3
n1 = math.pow(2, 3)
print(n1)# 8
ту ж саму операцію можна виконати так
n2 = 2**3
print(n2)
квадратний корінь числа
print(math.sqrt(9))# 3
# найближче найбільше ціле число
print(math.ceil(4.56))# 5
найближче найменше ціле число
print(math.floor(4.56))# 4
# Переведення з радіан в градуси
print(math.degrees(3.14159))# 180
# Переведення з градусів в радіани
print(math.radians(180))# 3.1415.....
# косинус
```

```
print(math.cos(math.radians(60)))# 0.5
# Синус
print(math.sin(math.radians(90)))# 1.0
# тангенс
print(math.tan(math.radians(0)))# 0.0
print(math.log(8,2))# 3.0
print(math.log10(100))# 2.0
```

Також модуль math надає ряд вбудованих констант, такі як PI та E:

```
import math
radius = 30
площа кола з радіусом 30
area = math.pi * math.pow (radius, 2)
print(area)
# натуральний логарифм числа 10
number = math.log(10, math.e)
print(number)
```

Додаткові математичні функції

Варто відзначити, що Python є ще ряд вбудованих функцій, які виконують деякі математичні обчислення, але не входять в модуль `math`. Зазначу деякі:

- abs : повертає абсолютне значення числа
- min : повертає мінімальне значення зі списку
- max : повертає максимальне значення зі списку

Наприклад, знайдемо "відстань" між двома числами (абсолютну різницю без урахування знака):

```
num1 = 3
num2 = 8
diff = abs (num1-num2)# 5
print(diff)# 5
```

Або знайдемо мінімальну та максимальну кількість у списку:

```
numbers = [54, 23, 1, 4, 657, 2, -3, 56, 24]
min_number = min(numbers)# -3
max_number = max(numbers)# 657
print("min:", min_number)
print("max:", max_number)
```

Модуль locale

При форматуванні чисел Python за замовчуванням використовує англосаксонську систему, при якій розряди цілого числа відокремлюються один від одного комами, а частина від цілої відокремлюється точкою. У континентальній Європі, наприклад, використовується інша система, при якій розряди розділяються точкою, а дробова та ціла частина - комою:

```
англосаксонська система
1,234.567
# європейська система
1.234,567
```

І для вирішення проблеми форматування під певну культуру в Python є вбудований модуль locale .

Для встановлення локальної культури в модулі locale визначено функцію setlocale() . Вона приймає два параметри:

```
setlocale(category, locale)
```

Перший параметр вказує на категорію, до якої застосовується функція - до чисел, валют або числах, і валют. Як значення для параметра ми можемо передавати одну з наступних констант:

- LC_ALL : застосовує локалізацію всім категоріям - до форматування чисел, валют, дат тощо.
- LC_NUMERIC : застосовує локалізацію до чисел
- LC_MONETARY : застосовує локалізацію до валют
- LC_TIME : застосовує локалізацію до дат та часу
- LC_CTYPE : застосовує локалізацію під час перекладу символів у верхній або нижній регістр
- LC_COLLIATE : застосовує локаль для порівняння рядків

Другий параметр функції setlocale вказує на локальну культуру, яку треба використовувати. На ОС Windows можна використовувати код країни ISO із двох символів, наприклад, для США - "us", для Німеччини - "de", для Росії - "ru". Але на MacOS необхідно вказувати код мови та код країни, наприклад, для англійської в США - "en_US", для німецької в Німеччині - "de_DE", для російської в Росії - "ru_RU". За промовчанням фактично використовується культура "en_US".

Безпосередньо для форматування чисел та валют модуль locale надає дві функції:

- `currency(num)` : форматує валюту

- `format_string(str, num)` : підставляє число num замість плейсхолдера у рядок str

  Застосовуються такі плейсхолдери:

  > - `d` : для цілих чисел
  > - `f` : для чисел з плаваючою точкою
  > - `e` : для експоненціального запису чисел

  Перед кожним плейсхолдером ставиться знак відсотка %, наприклад:

  ```
  "%d"
  ```

  При виведенні дробових чисел перед плейсхолдером після точки можна вказати скільки знаків у дробовій частині повинно відображатися:

  ```
  %.2f# два знаки в дрібній частині
  ```

Застосуємо локалізацію чисел та валют у німецькій культурі:

```
import locale
locale.setlocale(locale.LC_ALL, "de")# для Windows
# locale.setlocale(locale.LC_ALL, "de_DE")# для MacOS
number = 12345.6789
formatted = locale.format_string("%f", number)
print(formatted)# 12345,678900
formatted = locale.format_string("%.2f", number)
print(formatted)# 12345,68
formatted = locale.format_string("%d", number)
print(formatted)# 12345
formatted = locale.format_string("%e", number)
print(formatted)# 1,234568e+04
грошi = 234.678
formatted = locale.currency(money)
print(formatted)# 234,68 €
```

Якщо замість конкретного коду в якості другого параметра передається порожній рядок, Python буде використовувати культуру, яка застосовується на поточній робочій машині. А за допомогою функції getlocale() можна отримати цю культуру:

```
import locale
locale.setlocale(locale.LC_ALL, "")
number = 12345.6789
formatted = locale.format_string("%.02f", number)
print(formatted)# 12345,68
print(locale.getlocale())
```

```
# ('Russian_Russia', '1251') - Windows
# ('ru_RU', 'UTF-8') - MacOS
```

Залежно від системи висновок може відрізнятися.

Модуль decimal

При роботі з числами з плаваючою точкою (тобто float) ми стикаємося з тим, що в результаті обчислень ми отримуємо не зовсім правильний результат:

```
number = 0.1 + 0.1 + 0.1
print(number)# 0.30000000000000004
```

Проблему може вирішити використання функції round() , яка округлить число. Однак є інший спосіб, який полягає у використанні вбудованого модуля decimal .

Ключовим компонентом для роботи з числами цього модуля є клас Decimal . Для застосування нам треба створити його об'єкт за допомогою конструктора. Конструктор передається рядкове значення, яке представляє число:

```
від import decimal Decimal
number = Decimal("0.1")
```

Після цього об'єкт Decimal можна використовувати в арифметичних операціях:

```
від import decimal Decimal
number = Decimal("0.1")
number = number + number + number
print(number)# 0.3
```

В операціях з Decimal можна використовувати цілі числа:

```
number = Decimal("0.1")
number = number + 2
```

Однак не можна змішувати в операціях дробові числа float та Decimal:

```
number = Decimal("0.1")
number = number + 0.1# тут виникне помилка
```

За допомогою додаткових знаків ми можемо визначити, скільки символів буде в дробовій частині числа:

```
number = Decimal("0.10")
number = 3 * number
```

```
print(number)# 0.30
```

Рядок "0.10" визначає два знаки в дрібній частині, навіть якщо останні символи будуть представляти нуль. Відповідно "0.100" представляє три знаки в дрібній частині.

Округлення чисел

Об'єкти Decimal мають метод quantize() , що дозволяє округляти числа. Цей метод як перший аргумент передається також об'єкт Decimal, який вказує формат округлення числа:

```
від import decimal Decimal
number = Decimal("0.444")
number = number.quantize(Decimal("1.00"))
print(number)# 0.44
number = Decimal("0.555678")
print(number.quantize(Decimal("1.00")))# 0.56
number = Decimal("0.999")
print(number.quantize(Decimal("1.00")))# 1.00
```

Використовуваний рядок "1.00" зазначає, що округлення йтиме до двох знаків у дрібній частині.

За умовчанням округлення описується константою ROUND_HALF_EVEN , при якому округлення відбувається до найближчого парного числа, якщо округлена частина дорівнює 5. Наприклад:

```
від import decimal Decimal, ROUND_HALF_EVEN

number = Decimal("10.025")# 2 - найближче парне число
print(number.quantize(Decimal("1.00"), ROUND_HALF_EVEN))# 10.02
number = Decimal("10.035")# 4 - найближче парне число
print(number.quantize(Decimal("1.00"), ROUND_HALF_EVEN))# 10.04
```

Стратегія округлення передається як другий параметр quantize.

Рядок "1.00" означає, що округлення йтиме до двох чисел у дробовій частині. Але в першому випадку "10.025" – другим знаком йде 2 – парне число, тому, незважаючи на те, що наступне число 5, двійка не округляється до трійки.

У другому випадку "10.035" – другим знаком йде 3 – непарне число, найближчим парним числом буде 4, тому 35 округляється до 40.

Ця поведінка при округленні, можливо, не всім здасться бажаною, і в цьому випадку її можна перевизначити, використавши одну з наступних констант:

- ROUND_HALF_UP : округляє число у бік підвищення, якщо після нього йде число 5 або вище
- ROUND_HALF_DOWN : округляє число у бік підвищення, якщо після нього йде число більше 5

```
number = Decimal("10.026")
print(number.quantize(Decimal("1.00"), ROUND_HALF_DOWN))# 10.03
number = Decimal("10.025")
print(number.quantize(Decimal("1.00"), ROUND_HALF_DOWN))# 10.02
```

- ROUND_05UP : заокруглює 0 до одиниці, якщо після нього йде число 5 і вище

```
number = Decimal("10.005")
print(number.quantize(Decimal("1.00"), ROUND_05UP))# 10.01
number = Decimal("10.025")
print(number.quantize(Decimal("1.00"), ROUND_05UP))# 10.02
```

- ROUND_CEILING : округляє число у бік незалежно від того, яке число йде після нього

```
number = Decimal("10.021")
print(number.quantize(Decimal("1.00"), ROUND_CEILING))# 10.03
number = Decimal("10.025")
print(number.quantize(Decimal("1.00"), ROUND_CEILING))# 10.03
```

- ROUND_FLOOR : не округляє число незалежно від того, яке число йде після нього

```
number = Decimal("10.021")
print(number.quantize(Decimal("1.00"), ROUND_FLOOR))# 10.02
number = Decimal("10.025")
print(number.quantize(Decimal("1.00"), ROUND_FLOOR))# 10.02
```

Модуль dataclass. Data-класи
Модуль dataclasses надає декоратор dataclass , що дозволяє створювати data-класи - подібні дозволяють значно скоротити шаблонний код класів. Як правило, такі класи призначені для зберігання деякого стану, деяких даних і коли не потрібна якась поведінка у вигляді функцій.

Розглянемо найпростіший приклад:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
tom = Person("Tom", 38)
print(f"Name: {tom.name} Age: {tom.age}")# Name: Tom Age: 38
```

Тут визначено клас Person, у якого функції конструктора визначено два атрибути: name і age. Далі створюємо один об'єкт цього класу та виводимо значення його атрибутів на консоль.

Тепер змінимо цю програму, зробивши клас Person data-класом:

```
від dataclasses import dataclass
@dataclass
class Person:
    name: str
    age: int
tom = Person("Tom", 38)
print(f"Name: {tom.name} Age: {tom.age}")# Name: Tom Age: 38
```

Для створення data-класу імпортуємо з модуля dataclasses декоратор dataclass та застосовуємо його до класу Person. І в цьому випадку в самому класі нам уже не треба вказувати конструктор-функцію `__init__`. Ми просто вказуємо атрибути. А Python потім сам згенерує конструктор, який також ми можемо передати значення для атрибутів об'єкта.

Таким чином, ми вже скоротили визначення класу і зробили його простішим. Але генерацією методу `__init__` функціональність декоратора dataclass не обмежується. Насправді data-клас

```
@dataclass
class Person:
    name: str
    age: int
```

буде аналогічний наступному:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __repr__(self):
        return f"Person(name={self.name!r}, age={self.age!r}"

    def __eq__(self, other):
```

```
        if other.__class__ is self.__class__:
            return (self.name, self.age) == (other.name, other.age)
        return NotImplemented
```

В даному випадку ми бачимо, що крім функції __init__, також визначається функція `__repr__()` повернення рядкового подання і функція `__eq__()` для порівняння двох об'єктів. Застосування даних функцій:

```
від dataclasses import dataclass
@dataclass
class Person:
    name: str
    age: int

tom = Person("Tom", 38)
bob = Person("Bob", 42)
tomas = Person("Tom", 38)
print(tom == tomas)# True
print(tom == bob)# False
print(tom)# Person(name="Tom", age=38)
```

Параметри декоратора dataclass

За допомогою параметрів декоратор `dataclass` дозволяє згенерувати додатковий шаблонний код та взагалі налаштувати генерацію коду:

```
def dataclass(cls=None, /, *, init=True, repr=True, eq=True,
order=False,
              unsafe_hash=False, frozen=False, match_args=True,
              kw_only = False, slots = False)
```

Розглянемо базові параметри:

- init : якщо одно `True`, то генерується функція `__init__()`. За замовчуванням одно `True`
- repr : якщо `True`, то генерується функція `__repr__()`, яка повертає рядкове представлення об'єкта. За замовчуванням одно `True`
- eq : якщо `True`, то генерується функція `__eq__()`, яка порівнює два об'єкти. За замовчуванням одно `True`
- order : якщо `True`, то генеруються функції `__lt__` (операція <), `__le__` (<=), `__gt__` (>), `__ge__` (>=), які застосовуються для впорядкування об'єктів. За замовчуванням одно `False`
- unsafe_hash : якщо `True`, то генерується функція `__hash__()`, яка повертає хеш об'єкта. За замовчуванням одно `False`

Крім того, ті функції, які створюються за умовчанням, можуть бути перевизначені.

Застосування параметрів:

```
від dataclasses import dataclass
@dataclass(unsafe_hash=True, order=True)
class Person:
    name: str
    age: int
    def __repr__(self):
        return f"Person. Name: {self.name} Age: {self.age}"

tom = Person("Tom", 38)
print(tom.__hash__())# -421667297069596717
print(tom)# Person. Name: Tom Age: 38
```

В даному випадку включаємо генерування хешу та функцій упорядкування, а також явним чином перевизначаємо функцію `__repr__` для створення рядкового представлення об'єкта.

значення за замовчуванням

При необхідності атрибутам можна присвоїти значення за замовчуванням, якщо в конструкторі їм не передаються значення:

```
від dataclasses import dataclass
@dataclass
class Person:
    name: str
    age: int = 18

tom = Person("Tom", 38)
print(tom)# Person(name="Tom", age=38)
bob = Person("Bob")
print(bob)# Person(name="Bob", age=18)
```

Додавання додаткового функціоналу

Хоча data-класи призначені насамперед для зберігання різних даних, але також у них можна визначати поведінку за допомогою додаткових функцій:

```
від dataclasses import dataclass
@dataclass
class Person:
```

```
    name: str
    age: int
    def say_hello(self):
        print(f"{self.name} says hello")

tom = Person("Tom", 38)
tom.say_hello()# Tom says hello
```

Розділ 7. Рядки

Робота з рядками

Рядок представляє послідовність символів у кодуванні Unicode, укладених у лапки. Причому для визначення рядків Python дозволяє використовувати як одинарні, так і подвійні лапки:

```
message = "Hello World!"
print(message)# Hello World!
name = 'Tom'
print(name)# Tom
```

Якщо рядок довгий, його можна розбити на частини та розмістити на різних рядках коду. У цьому випадку весь рядок полягає в круглі дужки, а її окремі частини - у лапки:

```
text = ("Laudate omnes gentes laudate"
        "Magnificat in secula")
print(text)
```

Якщо ми хочемо визначити багаторядковий текст, такий текст полягає у потрійні подвійні чи одинарні лапки:

```
'''
Це коментар
'''
text = '''Laudate omnes gentes laudate
Magnificat in secula
Et anima mea laudate
Magnificat in secula
'''
print(text)
```

При використанні потрійних одинарних лапок не варто плутати їх із коментарями: якщо текст у потрійних одинарних лапках присвоюється змінним, то це рядок, а не коментар.

Керуючі послідовності у рядку

Рядок може містити ряд спеціальних символів - послідовностей, що управляють, або escape-послідовності. Деякі з них:

- \ : дозволяє додати всередину рядки слєш
- ' : дозволяє додати всередину рядка одинарну лапку
- " : дозволяє додати всередину рядка подвійну лапку
- \n : здійснює перехід на новий рядок
- \t : додає табуляцію (4 відступи)

Використовуємо деякі послідовності:

```
text = "Message:\n\"Hello World\""
print(text)
```

Консольний висновок програми:

```
Message:
"Hello World"
```

Хоча подібні послідовності можуть допомогти в деяких справах, наприклад, помістити в рядок лапку, зробити табуляцію, перенесення на інший рядок. Але вони також можуть заважати. Наприклад:

```
path = "C:\python\name.txt"
print(path)
```

Тут змінна path містить певний шлях до файлу. Однак усередині рядка зустрічаються символи "\n", які будуть інтерпретовані як послідовність, що управляє. Так, ми отримаємо наступний консольний висновок:

```
C:\python
ame.txt
```

Щоб уникнути подібної ситуації, перед рядком ставиться символ r

```
path = r"C:\python\name.txt"
print(path)
```

Вставка значень у рядок

Python дозволяє встроювати рядок значення інших змінних. Для цього всередині рядка змінні розміщуються у фігурних дужках {}, а перед усім рядком ставиться символ f :

```
userName = "Tom"
userAge = 37
user = f"name: {userName} age: {userAge}"
print(user)# name: Tom age: 37
```

У цьому випадку на місце `{userName}` вставлятиметься значення змінної userName. Аналогічно замість `{userAge}`
буде вставлятися значення змінної userAge.

Звернення до символів рядка

І ми можемо звернутися до окремих символів рядка за індексом у квадратних дужках:

```
string = "hello world"
c0 = string[0]# h
print(c0)
c6 = string[6]# w
print(c6)
c11 = string[11]# помилка IndexError: string index out of range
print(c11)
```

Індексація починається з нуля, тому перший символ рядка матиме індекс 0. А якщо ми спробуємо звернутися до індексу, якого немає в рядку, ми отримаємо виняток IndexError. Наприклад, у разі вище довжина рядка 11 символів, тому символи матимуть індекси від 0 до 10.

Щоб отримати доступ до символів, починаючи з кінця рядка, можна використовувати негативні індекси. Так, індекс -1 представлятиме останній символ, а -2 - передостанній символ і так далі:

```
string = "hello world"
c1 = string[-1]# d
print(c1)
c5 = string[-5]# w
print(c5)
```

Працюючи із символами слід враховувати, що рядок - це незмінний (immutable) тип, тому якщо спробуємо змінити якийсь окремий символ рядка, ми отримаємо помилку, як у наступному случае:

```
string = "hello world"
string[1] = "R"
```

Ми можемо тільки повністю перевстановити значення рядка, надавши їй інше значення.

Перебір рядка

За допомогою циклу for можна перебрати всі символи рядка:

```
string = "hello world"
for char in string:
    print(char)
```

Отримання підрядка

При необхідності ми можемо отримати з рядка не лише окремі символи, а й підрядок. Для цього використовується наступний синтаксис:

- `string[:end]` : витягується послідовність символів починаючи з 0-го індексу за індексом end (не включаючи)
- `string[start:end]` : вилучається послідовність символів починаючи з індексу start за індексом end (не включаючи)
- `string[start:end:step]` : вилучається послідовність символів починаючи з індексу start за індексом end (не включаючи) через крок step

Використовуємо всі варіанти отримання підрядка:

```
string = "hello world"
# з 0 до 5 індексу
sub_string1 = string[:5]
print(sub_string1)# hello
# з 2 до 5 індексу
sub_string2 = string[2:5]
print(sub_string2)# llo
# з 2 до 9 індексу через один символ
sub_string3 = string[2:9:2]
print(sub_string3)# lowr
```

Об'єднання рядків

Однією з найпоширеніших операцій із рядками є їхнє об'єднання чи конкатенація. Для об'єднання рядків застосовується операція додавання:

```
name = "Tom"
surname = "Smith"
fullname = name + " " + surname
print(fullname)# Tom Smith
```

З об'єднанням двох рядків все просто, але що, якщо нам треба скласти рядок і число? У цьому випадку необхідно привести число до рядка за допомогою функції
str() :

```
name = "Tom"
age = 33
info = "Name: " + name + " Age: " + str(age)
print(info)# Назва: Tom Age: 33
```

Повторення рядка

Для повторення рядка певну кількість разів застосовується операція множення:

```
print("a" * 3)# aaa
print("he" * 4)# hehehehe
```

Порівняння рядків

Особливо слід сказати порівняння рядків. Порівняння проводиться у лексикографічному порядку. При порівнянні рядків беруться до уваги символи та їх регістр. Так, цифровий символ умовно менше ніж будь-який алфавітний символ. Алфавітний символ у верхньому регістрі умовно менший, ніж алфавітні символи у нижньому регістрі. Наприклад:

```
str1 = "1a"
str2 = "aa"
str3 = "Aa"
print(str1 > str2)# False, тому що перший символ у str1 – цифра
print(str2 > str3)# True, тому що перший символ у str2 – у нижньому
регістрі
```

Тому рядок "1a" умовно менший, ніж рядок "aa". Спочатку порівняння йде за першим символом. Якщо початкові символи обох рядків є цифрами, то меншою вважається менша цифра, наприклад, "1a" менше, ніж "2a".

Якщо початкові символи представляють алфавітні символи у тому самому регістрі, то дивляться по алфавіту. Так, "aa" менше, ніж "ba", а "ba" менше, ніж

"ca".

Якщо перші символи однакові, до уваги беруться другі символи за наявності.

Залежність від регістру не завжди бажана, оскільки ми маємо справу з однаковими рядками. У цьому випадку перед порівнянням ми можемо привести обидва рядки до одного з регістрів.

Функція lower() наводить рядок до нижнього регістру, а функція upper() - до верхнього.

```
str1 = "Tom"
str2 = "tom"
print(str1 == str2)# False - рядки не рівні
print(str1.lower() == str2.lower())# True
```

Функції ord та len

Оскільки рядок містить символи Unicode, то за допомогою функції ord() ми можемо отримати числове значення для символу кодування Unicode:

```
print(ord("A"))# 65
```

Для отримання довжини рядка можна використовувати функцію len() :

```
string = "hello world"
length = len(string)
print(length)# 11
```

Пошук у рядку

За допомогою виразу `term in string` можна знайти підрядок term у рядку string. Якщо підрядок знайдено, то вираз поверне значення `True`, інакше повертається значення `False`:

```
text = "hello world"
exist = "hello" in text
print(exist)# True
exist = "sword" in text
print(exist)# False
```

Відповідно за допомогою операторів not in можна перевірити відсутність підрядки у рядку:

```
text = "hello world"
print("hello" not in text)# False
print("sword" not in text)# True
```

Додаткові матеріали

- Вправи для самоперевірки

Основні методи рядків
Розглянемо основні методи рядків, які ми можемо застосувати у додатках:

- isalpha() : повертає True, якщо рядок складається лише з алфавітних символів
- islower() : повертає True, якщо рядок складається лише із символів у нижньому регістрі
- isupper() : повертає True, якщо всі символи рядка у верхньому регістрі
- isdigit() : повертає True, якщо всі символи рядка – цифри
- isnumeric() : повертає True, якщо рядок є числом
- startswith(str) : повертає True, якщо рядок починається з підрядка str
- endswith(str) : повертає True, якщо рядок закінчується на підрядок str
- lower() : перекладає рядок у нижній регістр
- upper() : перекладає рядок у віховий регістр
- title() : початкові символи всіх слів у рядку перекладаються у верхній регістр
- capitalize() : перекладає у верхній регістр першу літеру тільки першого слова рядка
- lstrip() : видаляє початкові прогалини з рядка
- rstrip() : видаляє кінцеві пробіли з рядка
- strip() : видаляє початкові та кінцеві пробіли з рядка
- ljust(width) : якщо довжина рядка менша за параметр width, то праворуч від рядка додаються пробіли, щоб доповнити значення width, а сам рядок вирівнюється по лівому краю
- rjust(width) : якщо довжина рядка менша за параметр width, то ліворуч від рядка додаються пробіли, щоб доповнити значення width, а сам рядок вирівнюється праворуч.
- center(width) : якщо довжина рядка менша за параметр width, то ліворуч і праворуч від рядка рівномірно додаються пробіли, щоб доповнити значення width, а сам рядок вирівнюється по центру
- find(str[, start[, end]) : повертає індекс підрядка у рядку. Якщо підрядок не знайдено, повертається число -1
- replace(old, new[, num]) : замінює в рядку один підрядок на інший
- split([delimeter[, num]]) : розбиває рядок на підрядки в залежності від роздільника

- partition(delimeter) : розбиває рядок по роздільнику на три підрядки і повертає кортеж з трьох елементів - підрядок до роздільника, роздільник та підрядок після роздільника
- join(strs) : об'єднує рядки в один рядок, вставляючи між ними певний роздільник

Наприклад, якщо ми очікуємо введення з клавіатури числа, то перед перетворенням введеного рядка в число можна перевірити, чи за допомогою методу isnumeric() введено насправді число, і якщо так, то виконати операцію перетворення:

```
string = input("Введіть число: ")
if string.isnumeric():
    number = int(string)
    print(number)
```

Перевірка, починається або закінчується рядок на певний підрядок:

```
file_name = "hello.py"
starts_with_hello = file_name.startswith("hello")# True
ends_with_exe = file_name.endswith("exe")# False
```

Видалення пробілів на початку та в кінці рядка:

```
string = "hello world!"
string = string.strip()
print(string)# hello world!
```

Доповнення рядка пробілами та вирівнювання:

```
print("iPhone 7:", "52000".rjust(10))
print("Huawei P10:", "36000".rjust(10))
```

Консольний висновок:

```
iPhone 7: 52000
Huawei P10: 36000
```

Пошук у рядку

Для пошуку підрядки у рядку в Python застосовується метод find() , який повертає індекс першого входження підрядка в рядок і має три форми:

- `find(str)` : пошук підрядки str ведеться з початку рядка до кінця
- `find(str, start)` : параметр start задає початковий індекс, з якого буде здійснюватися пошук
- `find(str, start, end)` : параметр end визначає кінцевий індекс, до якого буде йти пошук

Якщо підрядок не знайдено, метод повертає -1:

```
welcome = "Hello world! Goodbye world!"
index = welcome.find("wor")
print(index)# 6
# пошук з 10-го індексу
index = welcome.find("wor",10)
print(index)# 21
# пошук з 10 по 15 індекс
index = welcome.find("wor",10,15)
print(index)# -1
```

Заміна у рядку

Для заміни в рядку одного підрядка на іншу застосовується метод replace() :

- `replace(old, new)` : замінює підрядок old на new
- `replace(old, new, num)` : параметр num вказує, скільки вхід підрядки old треба замінити на new. За замовчуванням num дорівнює -1, що відповідає першій версії методу та призводить до заміни всіх входжень.

```
phone = "+1-234-567-89-10"
# Заміна дефісів на пробіл
edited_phone = phone.replace("-", " ")
print(edited_phone)# +1 234 567 89 10
видалення дефісів
edited_phone = phone.replace("-", "")
print(edited_phone)# +12345678910
заміна тільки першого дефісу
edited_phone = phone.replace("-", "", 1)
print(edited_phone)# +1234-567-89-10
```

Поділ на підрядки

Метод split() розбиває рядок на список підрядків залежно від роздільника. Як роздільник може виступати будь-який символ або послідовність символів. Цей метод має такі форми:

- `split()` : як роздільник використовується пробіл
- `split(delimeter)` : як роздільник використовується delimeter
- `split(delimeter, num)` : параметр num вказує, скільки введень delimeter використовується для поділу. Частина рядка, що залишилася, додається до списку без поділу на підрядки.

```
text = "Це був величезний, у два обхвати дуб, з обламаними гілками та
з обламаною корою"
# поділ за пробілами
splitted_text = text.split()
print(splitted_text)
print(splitted_text[6])# дуб,
# розбиття по комах
splitted_text = text.split(",")
print(splitted_text)
print(splitted_text[1])# у два обхвати дуб
# розбиття по перших п'яти пробілах
splitted_text = text.split(" ", 5)
print(splitted_text)
print(splitted_text[5])# обхват дуб, з обламаними гілками і з
обламаною корою
```

Ще один метод - partition() розбиває рядок по роздільнику на три підрядки і повертає кортеж з трьох елементів - підрядок до роздільника, роздільник та підрядок після роздільника:

```
text = "Це був величезний, у два обхвати дуб, з обламаними гілками та
з обламаною корою"
text_parts = text.partition("дуб")
print(text_parts)
# ('Це був величезний, у два обхвати', 'дуб', ', з обламаними гілками
і з обламаною корою')
```

Якщо розділювач з рядку не знайдено, то повертається кортеж із одним рядком.

З'єднання рядків

Під час розгляду найпростіших операцій із рядками було показано, як поєднувати рядки з допомогою операції складання. Іншу можливість для з'єднання рядків є метод join() : він об'єднує список рядків. Причому поточний рядок, у якого викликається даний метод, використовується як роздільник:

```
words = ["Let", "me", "speak", "from", "my", "heart", "in",
"English"]
# роздільник - пропуск
```

```
sentence = " ".join(words)
print(sentence)# Let me speak from my heart in English
# Розділювач - вертикальна риса
sentence = " | ".join(words)
print(sentence)# Let | me | speak | від | my | heart | in | English
```

Замість списку метод join можна передати простий рядок, тоді розділювач буде вставлятися між символами цього рядка:

```
word = "hello"
joined_word = "|".join(word)
print(joined_word)# h|e|l|l|o
```

Додаткові матеріали

- Запитання для самоперевірки

Форматування
У минулих темах було розглянуто, як можна вставляти рядок деякі значення, передуючи рядок символом f :

```
first_name="Tom"
text = f"Hello, {first_name}."
print(text)# Hello, Tom.
name="Bob"
age=23
info = f"Name: {name}\t Age: {age}"
print(info)# Назви: Bob Age: 23
```

Але також у Python є альтернативний спосіб, який надає метод format() . Цей метод дозволяє форматувати рядок, вставляючи на місце плейсхолдерів певні значення.

Для вставки в рядок використовуються спеціальні параметри, які обрамляються дужками ({}).

Іменовані параметри

У рядку, що форматується, ми можемо визначати параметри, в методі `format()` передавати для цих параметрів значення:

```
text = "Hello, {first_name}.".format(first_name="Tom")
print(text)# Hello, Tom.
```

```
info = "Name: {name}\t Age: {age}".format(name="Bob", age=23)
print(info)# Назви: Bob Age: 23
```

Причому метод формат аргументи визначаються з тим самим ім'ям, що і параметри в рядку. Тож якщо параметр називається `first_name`, як у першому випадку, то аргумент, якому присвоюється значення, також називається `first_name`.

Параметри за позицією

Ми також можемо послідовно передавати в метод format набір аргументів, а в самому рядку, що форматується, вставляти ці аргументи, вказуючи у фігурних дужках їх номер (нумерація починається з нуля):

```
info = "Name: {0}\t Age: {1}".format("Bob", 23)
print(info)# Назви: Bob Age: 23
```

При цьому аргументи можна вставляти в рядок багато разів:

```
text = "Hello, {0} {0} {0}.".format("Tom")
```

Підстановки

Ще один спосіб передачі форматованих значень рядок представляє використання підстановок або спеціальних плейсхолдерів, на місце яких вставляються певні значення. Для форматування ми можемо використовувати такі плейсхолдери:

- s : для вставки рядків
- d : для вставлення цілих чисел
- f : для вставлення дробових чисел. Для цього типу також можна визначити через точку кількість знаків у дрібній частині.
- % : примножує значення на 100 і додає знак відсотка
- e : виводить число в експоненційному записі

Загальний синтаксис плейсхолдера наступний:

```
{:плейсхолдер}
```

Залежно від плейсхолдера можна додавати додаткові параметри. Наприклад, для форматування чисел float можна використовувати такі параметри

```
{:[кількість_символів][кома][.число_знаків_в_дрібної_частини]
плейсхолдер}
```

При виклику методу format у нього як аргументи передаються значення, які вставляються на місце плейсхолдерів:

```
welcome = "Hello {:s}"
name = "Tom"
formatted_welcome = welcome.format(name)
print(formatted_welcome)# Hello Tom
```

Як результат метод `format()` повертає новий відформатований рядок.

Форматування цілих чисел:

```
source = "{:d} символів"
number = 5
target = source.format(number)
print(target)# 5 символів
```

Якщо число, що форматується більше 999, то ми можемо вказати у визначенні плейсхолдера, що ми хочемо використовувати кому як роздільник розрядів:

```
source = "{:,d} символів"
print(source.format(5000))# 5,000 символів
```

Причому плейсхолдери можна використовувати і у f-рядках:

```
n = 5000
source = f"{n:,d} символів"
print(source)# 5,000 символів
```

Для дробових чисел, тобто таких, які представляють тип float, перед кодом плейсхолдера після точки можна вказати, скільки знаків у дробовій частині хочемо вивести:

```
number = 23.8589578
print("{:.2f}".format(number))# 23.86
print("{:.3f}".format(number))# 23.859
print("{:.4f}".format(number))# 23.8590
print("{:,.2f}".format(10001.23554))# 10,001.24
```

Ще один параметр дозволяє встановити мінімальну ширину форматованого значення символів:

```
print("{:10.2f}".format(23.8589578))# 23.86
print("{:8d}".format(25))# 25
```

Аналогічний приклад з f-рядками:

```
n1 = 23.8589578
print(f"{n1:10.2f}")# 23.86
n2 = 25
print(f"{n2:8d}")# 25
```

Для виведення відсотків краще скористатися кодом "%":

```
number = .12345
print("{:%}".format(number))# 12.345000%
print("{:.0%}".format(number))# 12%
print("{:.1%}".format(number))# 12.3%
print(f"{number:%}")# 12.345000%
print(f"{number:.0%}")# 12%
print(f"{number:.1%}")# 12.3%
```

Для виведення числа в експоненційному записі застосовується плейсхолдер "e":

```
number = 12345.6789
print("{:e}".format(number))# 1.234568e+04
print("{:.0e}".format(number))# 1e+04
print("{:.1e}".format(number))# 1.2e+04
print(f"{number:e}")# 1.234568e+04
print(f"{number:.0e}")# 1e+04
print(f"{number:.1e}")# 1.2e+04
```

Форматування без методу format

Існує також ще один спосіб форматування за допомогою наступного синтаксису:

```
рядок%(параметр1, параметр2,..параметрN)
```

Тобто на початку йде рядок, який містить ті ж плейсхолдери, які були розглянуті вище (за винятком плейсхолдера %), після рядка ставиться знак відсотка %, а потім список значень, що вставляються у рядок. Фактично знак відсотка є операцією, в результаті якої утворюється новий рядок:

```
info = "Ім'я: %s \t Вік: %d" % ("Tom", 35) Hello World !
print(info)# Ім'я: Tom Вік: 35
```

Поруч із плейсхолдером вказується знак відсотка і на відміну від функції format тут не потрібні фігурні дужки.

Причому способи форматування чисел тут також застосовуються:

```
number = 23.8589578
print("%0.2f - %e" % (number, number))# 23.86 - 2.385896e+01
```

Встановлення довжини рядка

За допомогою спеціальних символів можна задати довжину рядка під час форматування:

- <N : вирівнює рядок по лівому краю та доповнює його пробілами з правого боку до довжини N
- > N : вирівнює рядок з правого краю і доповнює його пробілами з лівого боку до довжини N
- ^N : вирівнює рядок по центру і доповнює його пробілами з лівого та правого боку до довжини N
- .N : задає точну довжину рядка. Якщо в ній більше N символів, то вона усікається

Наприклад:

```
str = "Hello World"
print(f"{str:>16}!")
print(f"{str:<16}!")
print(f"{str:^16}!")
print(f"{str:.16}!")
print(f"{str:.5}!")
```

Результат:

```
    Hello World!
Hello World!
  Hello World!
Hello World!
Hello!
```

Розділ 8. Pattern matching

Конструкція match

Починаючи з версії 3.10, у мові Python з'явилася така функціональність як pattern matching (порівняння шаблонів). Pattern matching представляє застосування конструкції match , яка дозволяє зіставити вираз з деяким шаблоном. І якщо вираз відповідає шаблону, виконуються певні дії. У цьому сенсі конструкція match схожа на конструкцію if/else/elif , яка виконує певні дії в залежності від певної умови. Однак функціональність match набагато ширша - вона також дозволяє витягти дані зі складових типів та застосувати дії до різних частин об'єктів.

Конструкція match має таке формальне визначення:

```
match вираз:
    case шаблон_1:
        дію_1
    case шаблон_2:
        дію_2
    ................
    case шаблон_N:
        дію_N
    case _:
        дія_за замовчуванням
```

Після ключового слова match йде порівнюваний вираз. І потім після двокрапки на наступних рядках розташовуються вирази case . Після кожного виразу case вказується шаблон, з яким порівнюється вираз match. Після шаблону через двокрапку вказуються набір дій блоку case, що виконуються.

Конструкція match послідовно порівнює вираз із шаблонами з блоків case. І якщо був знайдений шаблон з якогось блоку case відповідає виразу з match, то виконуються інструкції з даного блоку case.

В якості патернів/шаблонів, з якими порівнюються вирази, можуть застосовуватися дані примітивних типів, так і послідовності елементів і об'єктів класів.

Спочатку розглянемо ситуацію, коли як шаблон виступають літерали примітивних типів. Наприклад, залежно від мови виведемо вітальне повідомлення:

```
def print_hello(language):
    match language:
        case "російський":
            print("Привіт")
        case "english":
            print("Hello")
        case "german":
            print("Hallo")
```

```
print_hello("english")# Hello
print_hello("german")# Hallo
print_hello("російський")# Привіт
```

Тут функція `print_hello` приймає параметр language, через який передається вибрану мову. У самій функції конструкція match порівнює значення змінної language. У блоках case визначаються шаблони - рядки, із якими зіставляється змінна language.

Наприклад, при виклики `print_hello("english")` параметр language дорівнює "english", тому конструкція match вибере наступний блок case:

```
case "english":
    print("Hello")
```

Зверніть увагу, що блоки case мають відступи від початку конструкції match. Інструкції кожного блоку case мають відступи від початку даного блоку case. Але якщо блок case має одну інстукцію, її можна помістити на тому ж рядку, що оператор case:

```
def print_hello(language):
    match language:
        case "russian": print("Привіт")
        case "english": print("Hello")
        case "german": print("Hallo")

print_hello("english")# Hello
print_hello("german")# Hallo
print_hello("російський")# Привіт
```

Причому якщо вираз з match не відповідає жодному шаблону case, то відповідно жоден з цих блоків case не виконується.

Якщо необхідно, щоб при розбіжності значень (якщо жоден із шаблонів case не відповідає виразу match) виконувались деякі дії за умовчанням, то в цьому випадку застосовується шаблон _ (прочерк):

```
def print_hello(language):
    match language:
        case "російський":
            print("Привіт")
        case "english":
            print("Hello")
        case "german":
            print("Hallo")
```

```
        case _:
            print("Undefined")

print_hello("english")# Hello
print_hello("spanish")# Undefined
```

Якщо жоден із шаблонів case не відповідає значенню language, то виконуватиметься блок:

```
case _:
    print("Undefined")
```

Але також можна визначити блок case, який дозволяє порівнювати з кількома знеченями. У цьому випадку значення поділяються вертикальною рисою:

```
def print_hello(language):
    match language:
        case "російський":
            print("Привіт")
        case "american english" | "british english" | "english":
            print("Hello")
        case _:
            print("Undefined")

print_hello("english")# Hello
print_hello("american english")# Hello
print_hello("spanish")# Undefined
```

У разі шаблон `case "american english" | "british english" | "english"` відповідає відразу трьом значенням.

Так само можна порівнювати вирази з даними інших типів. Наприклад:

```
def operation(a, b, code):
    match code:
        case 1:
            return a + b
        case 2:
            return a - b
        case 3:
            return a * b
        case _:
            return 0

print(operation(10, 5, 1))# 15
print(operation(10, 5, 2))# 5
```

```
print(operation(10, 5, 3))# 50
print(operation(10, 5, 4))# 0
```

Тут функція operation приймає два числа та код операції. Конструкція match порівнює код операції з конкретними значеннями та залежно від значення виконує на числами певну операцію. Наприклад, якщо code дорівнює 1, то виконується вираз:

```
case 1:
    return a + b
```

цей вираз case поверне з функцію суму чисел a та b.

Аналогічно якщо code = 2, то повертається різниця, і якщо code = 3, то повертається добуток чисел. У решті випадків повертається 0.

Кортежі в pattern matching
Як шаблони в pathern matching в Python можуть виступати кортежі. Наприклад:

```
def print_data(user):
    match user:
        case ("Tom", 37):
            print("default user")
        case ("Tom", age):
            print(f"Age: {age}")
        case (name, 22):
            print(f"Name: {name}")
        case (name, age):
            print(f"Name: {name} Age: {age}")

print_data(("Tom", 37))# default user
print_data(("Tom", 28))# Age: 28
print_data(("Sam", 22))# Name: Sam
print_data(("Bob", 41))# Name: Bob Age: 41
print_data(("Tom", 33, "Google"))# не відповідає жодному з шаблонів
```

В даному випадку функція приймає параметр user, який, як передбачається, представляє кортеж із двох елементів. І конструкція match порівнює цей кортеж із рядом шаблонів. Перший шаблон передбачає, що кортеж user точно відповідає набору значень:

```
case ("Tom", 37):
    print("default user")
```

Тобто якщо перший елемент кортежу дорівнює "Tom", а другий - 37, то на консоль виводиться рядок "default user"

Другий шаблон відповідає будь-якому двоелементному кортежу, перший елемент якого дорівнює рядку "Tom":

```
case ("Tom", age):
    print(f"Age: {age}")
```

Для другого елемента визначається змінна age. У результаті, якщо перший елемент кортежу дорівнює рядку "Tom", а другий не дорівнює 37, такий кортеж буде відповідати другому шаблону. Причому другий елемент передаватиметься змінною age.

Третій шаблон багато в чому аналогічний, тільки тепер суворо визначений другий елемент кортежу - він повинен дорівнювати 22, а перший потрапляє в змінну name:

```
case (name, 22):
    print(f"Name: {name}")
```

Якщо двоелементний кортеж відповідає першому, другому і третьому шаблонам, він відповідатиме четвертому шаблону, у якому нам не важливі конкретні значення - їм визначені змінні name і age:

```
case (name, age):
    print(f"Name: {name} Age: {age}")
```

Альтернативні значення

Якщо необхідно, щоб елемент кортежу відповідав набору значень, ці значення можна перерахувати через вертикальну межу:

```
def print_data(user):
    match user:
        case ("Tom" | "Tomas" | "Tommy", 37):
            print("default user")
        case ("Tom", age):
            print(f"Age: {age}")
        case (name, 22):
            print(f"Name: {name}")
        case (name, age):
            print(f"Name: {name} Age: {age}")
```

```
print_data(("Tom", 37))# default user
print_data(("Tomas", 37))# default user
print_data(("Tom", 28))# Age: 28
print_data(("Sam", 37))# Name: Sam Age: 37
```

У цьому випадку перший шаблон відповідає двоелементному кортежу, де перший елемент дорівнює або Tom, або Tomas, або Tommy.

Також можна задати альтернативні значення для окремих елементів, але й альтернативні кортежі:

```
def print_data(user):
    match user:
        case ("Tom", 37) | ("Sam", 22):
            print("default user")
        case (name, age):
            print(f"Name: {name} Age: {age}")

print_data(("Tom", 37))# default user
print_data(("Sam", 22))# default user
print_data(("Mike", 28))# Name: Mike Age: 28
```

У цьому випадку перший шаблон відповідатиме двом кортежам: `("Tom", 37)` і `("Sam", 22)`

Пропуск елементів

Якщо нам не важливий елемент кортежу, то в шаблоні замість конкретного значення або змінної можна вказати шаблон _:

```
def print_data(user):
    match user:
        case ("Tom", 37):
            print("default user")
        case (name, _): другий елемент не важливий
            print(f"Name: {name}")

print_data(("Tom", 37))# default user
print_data(("Sam", 25))# Name: Sam
print_data(("Bob", 41))# Name: Bob
```

Можна використовувати прочерки для всіх елементів кортежу, у цьому випадку значення всіх цих елементів будуть не важливими:

```
def print_data(user):
    match user:
```

```
        case ("Tom", 37):
            print("default user")
        case ("Sam", _):
            print("Name: Sam")
        case (_, _):
            print("Undefined user")

print_data(("Tom", 37))# default user
print_data(("Sam", 25))# Name: Sam
print_data(("Bob", 41))# Undefined user
```

Причому в останньому випадку шаблон, як і `(_, _)` раніше, відповідає тільки двоелементному кортежу.

У прикладі вище застосовувані шаблони відповідали лише двоелементного кортежу. Однак також можна використовувати одночасно шаблони кортежів з різною кількістю елементів:

```
def print_data(user):
    match user:
        case (name, age):
            print(f"Name: {name} Age: {age}")
        case (name, age, company):
            print(f"Name: {name} Age: {age} Company: {company}")
        case (name, age, company, lang):
            print(f"Name: {name} Age: {age} Company: {company}
Language: {lang}")

print_data(("Tom", 37))# Name: Tom Age: 37
print_data(("Sam", 22, "Microsoft"))# Name: Sam Age: 22 Company:
Microsoft
print_data(("Bob", 41, "Google", "english"))
# Name: Bob Age: 41 Company: Google Language: english
```

Кортеж з невизначеною кількістю елементів

Якщо необхідно порівнювати вираз із кортежем невизначеної довжини, то можна визначати всі інші значення кортежу за допомогою символу * (зірочки):

```
def print_data(user):
    match user:
        case ("Tom", 37, *rest):
            print(f"Rest: {rest}")
        case (name, age, *rest):
            print(f"{name} ({age}): {rest}")
```

```
print_data(("Tom", 37))# Rest: []
print_data(("Tom", 37, "Google"))# Rest: ["Google"]
print_data(("Bob", 41, "Microsoft", "english"))# Bob (41):
["Microsoft", "english"]
```

У прикладі вище застосовується параметр `*rest`, який відповідає решті всіх елементів. Тобто, у прикладі вище шаблони `("Tom", 37, *rest)` і `(name, age, *rest)` відповідають будь-якому кортежу з двома елементами і більше. Всі елементи, починаючи з третього, будуть поміщатися в параметр rest, який представляє масив значень.

Якщо нам цей параметр (rest) не важливий, але ми, як і раніше, хочемо, щоб шаблон відповідав кортежу з невизначеною кількістю елементів, ми можемо використовувати підшаблон `*_`:

```
def print_data(user):
    match user:
        case ("Tom", 37, *_):
            print("Default user")
        case (name, age, *_):
            print(f"{name} ({age})")

print_data(("Tom", 37))# Default user
print_data(("Tom", 37, "Google"))# Default user
print_data(("Bob", 41, "Microsoft", "english"))# Bob (41)
```

Массивы в pattern matching

В качестве шаблонов также могут выступать массивы. Подобным шаблоны также могут содержать либо конкретные значения, либо переменные, которые передаются элементы
массивов, либо символ прочерка _, если элемент массива не важен:

```
def print_people(people):
    match people:
        case ["Tom", "Sam", "Bob"]:
            print("default people")
        case ["Tom", second, _]:
            print(f"Second Person: {second}")
        case [first, second, third]:
            print(f"{first}, {second}, {third}")


print_people(["Tom", "Sam", "Bob"])        # default people
print_people(["Tom", "Mike", "Bob"])       # Second Person: Mike
print_people(["Alice", "Bill", "Kate"])    # Alice, Bill, Kate
```

```
print_people(["Tom", "Kate"])              # несоответствует ни
одному из шаблонов
```

В данном случае функция print_people принимает массив, который, как предполагается, состоит из трех элементов.

Первый шаблон предполагает, что элементы массива имеют определенные значения:

```
case ["Tom", "Sam", "Bob"]:
    print("default people")
```

В данном случае первый элемент массива должен представлять строку "Tom", второй - строку "Sam" и третий - строку "Bob".

Второй шаблон предполагает, что первый элемент массива должен быть равне строке "Tom", остальные два элемента могут иметь произвольные значения:

```
case ["Tom", second, _]:
    print(f"Second Person: {second}")
```

При этом значение второго элемента передается в переменную second, а значение третьего элемента не важно, поэтому вместо него применяется прочерк.

Третий шаблон соответствует любому массиву из трех элементов. При этом его элементы передаются в переменные first, second и third:

```
case [first, second, third]:
    print(f"{first}, {second}, {third}")
```

В данном случае для соответствия любому из шаблонов массив должен был иметь три элемента. Но также можно определять шаблоны для массивов разной длины:

```
def print_people(people):
    match people:
        case [_]:
            print("Массив из одного элемента")
        case [_, _]:
            print("Массив из двух элементов")
        case [_, _, _]:
            print("Массив из трех элементов")
        case _:
```

```
            print("Непонятно")


print_people(["Tom"])                   # Массив из одного элемента
print_people(["Tom", "Sam"])            # Массив из двух элементов
print_people(["Tom", "Sam", "Bob"])     # Массив из трех элементов
print_people("Tom")                     # Непонятно
```

Массивы неопределенной длины

Если необходимо сравнивать выражение с массивом неопределенной длины, то можно определить значения/переменные только для обязательных элементов массива, а на необязательные ссылаться с помощью символа * (звездочки):

```
def print_people(people):
    match people:
        case [first, *other]:
            print(f"First: {first}  Other: {other}")


print_people(["Tom"])                   # First: Tom  Other: []
print_people(["Tom", "Sam"])            # First: Tom  Other: ["Sam"]
print_people(["Tom", "Sam", "Bob"])     # First: Tom  Other: ["Sam",
"Bob"]
```

В примере выше применяется параметр `*other`, который соответствует всем остальным элементам.
То есть шаблон `[first, *other]` соответствует любому массиву, который имеет как минимум один элемент, и этот элемент будет помещаться в параметр first. Все остальные элементы помещаются в параметр other, который представляет массив значений.

Если нам параметр с символом * (other) не важен, но мы по прежнему хотим, чтобы шаблон соответствовал массиву с одним и большим количеством элементов,
мы можем использовать подшаблон `*_`:

```
def print_people(people):
    match people:
        case [first, *_]:
            print(f"First: {first}")


print_people(["Tom"])                   # First: Tom
```

```
print_people(["Sam", "Tom"])           # First: Sam
print_people(["Bob", "Sam", "Tom"])    # First: Bob
```

Альтернативные значения

Если необходимо, чтобы элемент массива соответствовал набору значений, то эти значения можно перечислить через вертикальную черту:

```
def print_people(people):
    match people:
        case ["Tom" | "Tomas" | "Tommy", "Sam", "Bob"]:
            print("default people")
        case [first, second, third]:
            print(f"{first}, {second}, {third}")


print_people(["Tom", "Sam", "Bob"])        # default people
print_people(["Tomas", "Sam", "Bob"])      # default people
print_people(["Alice", "Bill", "Kate"])    # Alice, Bill, Kate
```

В данном случае первый шаблон соответствует массиву из трех элементов, где первый элемент равен или "Tom", или "Tomas", или "Tommy".

Также можно задать альтернативные значения для отдельных элементов, но и альтернативные массивы:

```
def print_people(people):
    match people:
        case ["Tom", "Sam", "Bob"] | ["Tomas", "Sam", "Bob"]:
            print("Tom/Tomas, Sam, Bob")
        case [first, second, third]:
            print(f"{first}, {second}, {third}")


print_people(["Tom", "Sam", "Bob"])        # Tom/Tomas, Sam, Bob
print_people(["Tomas", "Sam", "Bob"])      # Tom/Tomas, Sam, Bob
print_people(["Alice", "Bill", "Kate"])    # Alice, Bill, Kate
```

В данном случае первый шаблон будет соответствовать двум массивам: `["Tom", "Sam", "Bob"]` и `["Tomas", "Sam", "Bob"]`

Словники в pattern matching
Pattern matching дозволяє перевірити наявність у словнику певних ключів та значень:

```
def look(words):
    match words:
        case {"red": "червоний", "blue": "синій"}:# якщо у словнику
words слова red і blue
            print("Слова red і blue є у словнику")
        case {"red": "червоний"}:# якщо у словнику words є слово red
            print("Слово red є у словнику, а слово blue відсутнє")
        case {"blue": "синій"}:# якщо у словнику words є слово blue
            print("Слово blue є у словнику, а слово red відсутнє")
        case {}:
            print("Слова red і blue у словнику відсутня")
        case _:
            print("Це не словник")

look({"red": "червоний", "blue": "синій", "green": "зелений"})# Слова
red і blue є у словнику
look({"red": "червоний", "green": "зелений"})# Слово red є у
словнику, а слово blue відсутнє
look({"blue": "синій", "green": "зелений"})# Слово blue є у словнику,
а слово red відсутнє
look({"green": "зелений"})# Слова red і blue у словнику відсутня
look("yelllow")# Це не словник
```

Тут передбачається, що у функцію look передається словник. Перший шаблон

```
case {"red": "червоний", "blue": "синій"}:# якщо у словнику words
слова red і blue
    print("Слова red і blue є у словнику")
```

відповідає словнику, в якому є два елементи з наступними ключами та значеннями: `"red": "красный"` і `"blue": "синий"`.

Другий шаблон (`{"red": "красный"}`) відповідає будь-якому словнику, де є елемент `"red": "красный"`. Аналогічно третій шаблон (`{"blue": "синий"}`) відповідає будь-якому словнику, де є елемент `"blue": "синий"`

Четвертий шаблон `case {}` відповідає в принципі будь-якому словнику.

Останній шаблон відповідає будь-якому значенню і застосовується на випадок, якщо функцію передано не словник.

Передача набору значень

За допомогою вертикальної межі | можна визначити альтернативні значення:

```
def look(words):
    match words:
        case {"red": "червоний" | "червоний" | "червоний"}:# якщо
значення "червоний", "червоний" або "червоний"
            print("Слово red є у словнику")
        case {}:
            print("Слово red у словнику відсутнє або має некоректне
значення")

look({"red": "червоний", "green": "зелений"})# Слово red є у словнику
look({"red": "червоний", "green": "зелений"})# Слово red є у словнику
look({"green": "зелений"})# Слово red у словнику відсутнє або має
некоректне значення
```

В даному випадку шаблон `{"red": "красный" | "алый" | "червонный"}` відповідає словнику, в якому є елемент з ключем "red" та значенням "червоний" або "червоний" або "червоний".

Також можна задати альтернативний набір словників:

```
def look(words):
    match words:
        case {"red": "червоний"} | {"blue": "синій"} :
            print("або red, або blue є в словнику")
        case {}:
            print("треба перевірити слова red і blue")

look({"red": "червоний", "green": "зелений"})# або red, або blue є у
словнику
look({"blue": "синій", "green": "зелений"})# або red, або blue є у
словнику
look({"green": "зелений"})# треба перевірити слова red і blue
```

Перший шаблон - `{"red": "красный"} | {"blue": "синий"}` відповідає словнику, в якому є або елемент `{"red": "красный"}`, або `{"blue": "синий"}` обидва.

Якщо нам важливі самі ключі, але важливо ключів, то замість конкретних значень можна передати шаблон _:

```
def look(words):
    match words:
        case {"red": _, "blue": _}:
            print("Слова red і blue є у словнику")
        case {}:
            print("red та/або blue відсутні у словнику")
```

```
look({"red": "червоний", "blue": "синій"})# Слова red і blue є у словнику
look({"red": "червоний", "blue": "синій"})# Слова red і blue є у словнику
look({"red": "червоний", "green": "зелений"})# red та/або blue відсутні у словнику
```

Отримання значень за ключами

Pattern matching дозволяє отримати значення елементів змінні у вигляді:

```
{ключ: змінна}
```

Наприклад:

```
def look(words):
    match words:
        case {"red": red, "blue": blue}:
            print(f"red: {red} blue: {blue}")
        case {}:
            print("треба перевірити слова red і blue")

look({"red": "червоний", "blue": "синій"})# red: червоний blue: синій
look({"red": "червоний", "blue": "синій"})# red: червоний blue: синій
```

У першому шаблоні значення елемента з ключем "red" попадає в змінну red, а елемента з ключем "blue" - змінну blue.

Набуття всіх значень

За допомогою символів ** (подвійна зірочка) можна отримати інші елементи словника:

```
def look(words):
    match words:
        case {"red": red, **rest}:
            print(f"red: {red}")
            for key in rest:# rest - теж словник
                print(f"{key}: {rest[key]}")

look({"red": "червоний", "blue": "синій", "green": "зелений"})
# red: червоний
# blue: синій
# green: зелений
```

Тут шаблон `{"red": red, **rest}` відповідає будь-якому словнику, в якому є елемент з ключем "red". Всі інші елементи словника поміщаються в параметр rest, який у свою чергу представляє словник.

Класи в pattern matching
Python дозволяє використовувати в pattern matching як шаблони об'єкти класів. Розглянемо з прикладу:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

def print_person(person):
    match person:
        case Person(name="Tom", age=37):
            print("Default Person")
        case Person(name=name, age=37):
            print(f"Name: {name}")
        case Person(name="Tom", age=age):
            print(f"Age: {age}")
        Person(name=name, age=age):
            print(f"Name: {name} Age: {age}")

print_person(Person("Tom", 37))# Default person
print_person(Person("Tom", 22))# Age: 22
print_person(Person("Sam", 37))# Name: Sam
print_person(Person("Bob", 41))# Name: Bob Age: 41
```

Тут визначено клас Person, який через конструктор набуває значень для атрибутів `self.name` і `self.age`.

Функція print_person приймає параметр Person, який, як передбачається, є об'єктом класу Person. І всередині функції конструкція
match порівнює значення параметра person із рядом шаблонів. Кожен шаблон є визначення Person, де з кожним атрибутом зіставляється деяке значення. Наприклад, перший шаблон строго визначає значення обох атрибутів:

```
case Person(name="Tom", age=37):
    print("Default Person")
```

Даний шаблон відповідає об'єкту Person, якщо цей об'єкт має атрибут name значення "Tom", а атрибут age - значення 37.

Варто відзначити, що цей шаблон - це не виклик конструктора Person. Шаблон просто встановлює, як атрибути порівнюються зі значеннями.

Другий шаблон суворо задає значення лише для атрибуту age:

```
case Person(name=name, age=37):
    print(f"Name: {name}")
```

Для відповідності цьому шаблону атрибут age повинен дорівнювати 37. А атрибут name може мати довільне значення. І це значення передається змінною name. А запис `name=name` розшифровується як `атрибут_объекта=переменная`. А у виклику
`print(f"Name: {name}")` на консоль виводиться значення змінної name, яка набула значення атрибута name.

В даному випадку і атрибут, і змінна мають однакове значення, але це необов'язково, і для змінної можна використовувати інше значення, наприклад:

```
case Person(name=person_name, age=37):# змінною person_name
передається значення атрибута name
    print(f"Name: {person_name}")
```

Третій шаблон відповідає об'єкту Person, у якого атрибут name дорівнює рядку "Tom". А значення атрибуту age передається в змінну age:

```
case Person(name="Tom", age=age):
    print(f"Age: {age}")
```

І в останньому шаблоні атрибути name та age можуть мати довільні значення. І ці значення передаються однойменним змінним:

```
Person(name=name, age=age):
    print(f"Name: {name} Age: {age}")
```

При цьому нам необов'язково використовувати всі атрибути об'єкта Person. Також ми можемо застосувати патерн _, якщо нам потрібно обробити випадки, які не відповідають жодному шаблону:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

def print_person(person):
    match person:
        Person(name="Tom"):
            print("Default Person")
        case Person(name=person_name):# отримуємо лише атрибут name
```

```
            print(f"Name: {person_name}")
        case _:
            print("Not a Person")

print_person(Person("Tom", 37))# Default person
print_person(Person("Sam", 37))# Name: Sam
print_person("Tom")# Not a Person
```

У цьому випадку другий шаблон `Person(name=person_name)` відповідає будь-якому об'єкту Person, при цьому значення атрибута name передається змінною person_name

А останній шаблон обробляє випадки, коли передано значення, яке не становить об'єкт Person.

Передача набору значень

Також за допомогою вертикальної межі можна визначити набір значень, які повинен мати атрибут:

```
def print_person(person):
    match person:
        Person(name="Tom" | "Tomas" | "Tommy"):
            print("Default Person")
        case Person(name=person_name):# отримуємо лише атрибут name
            print(f"Name: {person_name}")
        case _:
            print("Not a Person")

print_person(Person("Tom", 37))# Default person
print_person(Person("Tomas", 37))# Default person
```

У цьому випадку перший шаблон відповідає об'єкту Person, у якого атрибут name має одне із трьох значень: "Tom", "Tomas" або "Tommy".

Також можна задавати альтернативні значення для всього шаблону, у тому числі за допомогою об'єктів інших класів:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

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

```
def print_person(person):
    match person:
        Case Person(name="Tom") | Student(name="Tomas"):
            print("Default Person/Student")
        Case Person(name=name) | Student(name=name):# отримуємо лише
атрибут name
            print(f"Name: {name}")
        case _:
            print("Not a Person or Student")

print_person(Person("Tom", 37))# Default Person/Student
print_person(Student("Tomas"))# Default Person/Student

print_person(Person("Bob", 41))# Name: Bob
print_person(Student("Mike"))# Name: Mike
print_person("Tom")# Not a Person or Student
```

Тут перший шаблон

```
Case Person(name="Tom") | Student(name="Tomas")
```

відповідає будь-якому об'єкту Person, у якого атрибут name = "Tom, та будь-якому об'єкту Student, у якого атрибут name = "Tomas".

Другий шаблон - `case Person(name=name) | Student(name=name)` відповідає будь-якому об'єкту Person та Student.

Позиційні параметри

У прикладах вище визначення атрибутів прописувалося їх ім'я: `case Person(name="Tom", age=37)`. Але якщо використовується купа шаблонів, і в кожному необхідно зв'язати атрибути об'єкта з деякими значеннями або змінними, то постійна згадка атрибутів можна дещо роздмухати код. Але Python також дозволяє використовувати позиційні параметри:

```
class Person:
    __match_args__ = ("name", "age")
    def __init__(self, name, age):
        self.name = name
        self.age = age

def print_person(person):
    match person:
        case Person("Tom", 37):
            print("Default Person")
```

```
        case Person(person_name, 37):
            print(f"Name: {person_name}")
        case Person("Tom", person_age):
            print(f"Age: {person_age}")
        case Person(person_name, person_age):
            print(f"Name: {person_name} Age: {person_age}")

print_person(Person("Tom", 37))# Default person
print_person(Person("Tom", 22))# Age: 22
print_person(Person("Sam", 37))# Name: Sam
print_person(Person("Bob", 41))# Name: Bob Age: 41
```

Зверніть увагу у класі Person на виклик функції:

```
__match_args__ = ("name", "age")
```

Завдяки цьому Python знатиме, що при вказівці атрибутів атрибут name буде йти першим, а атрибут age - другим.

І таким чином, шаблони не повинні вказувати ім'я атрибута: `case Person("Tom", 37)` - Python сам зіставить атрибути і значення/змінні на основі їх позиції.

guards або обмеження шаблонів
Guard або обмеження шаблонів дозволяють встановити додаткові умови, яким має відповідати вираз. Обмеження визначається відразу після шаблону за допомогою ключового слова if , після якого йде умова обмеження:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

def enter(person):
    match person:
        case Person(name=name, age=age) if age < 18:
            print(f"{name}, доступ заборонено")
        Person(name=name):
            print(f"{name}, ласкаво просимо!")

enter(Person("Tom", 37))# Tom, ласкаво просимо!
enter(Person("Sam", 12))# Sam, доступ заборонено
```

Тут перший шаблон

```
case Person(name=name, age=age) if age < 18:
    print(f"{name}, доступ заборонено")
```

Відповідає будь-якому об'єкту Person, у якого атрибут age менше 18. Власне частина `if age < 18` і становить обмеження. Відповідно, якщо у користувача вік менше 18, то виводиться одне повідомлення, якщо більше 18, то інше.

Подібним чином можна вводити додаткові обмеження:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

def enter(person):
    match person:
        case Person(name=name, age=age) if age < 18:
            print(f"{name}, доступ заборонено")
        case Person(name=name, age=age) if age < 22:
            print(f"{name}, доступ обмежений")
        Person(name=name):
            print(f"{name}, у вас повноцінний доступ!")

enter(Person("Tom", 37))# Tom, у вас повноцінний доступ!
enter(Person("Bob", 20))# Bob, доступ обмежений
enter(Person("Sam", 12))# Sam, доступ заборонено
```

Умови обмежень можуть бути складнішими і складнішими за структурою:

```
def check_data(data):
    match data:
        case name, age if name == "admin" або age not in range(1,
101):
            print("Некоректні значення")
        case name, age:
            print(f"Дані перевірені. Name: {name} Age: {age}")

check_data(("admin", -45))# Некоректні значення
check_data(("Tom", 37))# Дані перевірені. Name: Tom Age: 37
```

У цьому випадку функція одержує кортеж data. Обидва шаблони конструкції match відповідають двоелементному кортежу. Але перший шаблон також застосовує обмеження `name == "admin" or age not in range(1, 101)`, відповідно до якого перший елемент кортежу повинен мати значення "admin", а другий повинен знаходитись поза діапазоном 1-101.

Установка псевдонимов и паттерн AS
Оператор as позволяет установить псевдоним для значения шаблона или для всего шаблона. Простейший пример:

```
def print_person(person):
    match person:
        case "Tom" | "Tomas" | "Tommy" as name:
            print(f"Name: {name}")
        case _:
            print("Undefined")


print_person("Tom")    # Name: Tom
print_person("Tomas")  # Name: Tomas
print_person("Bob")    # Undefined
```

Здесь первый шаблон соответствует трем строкам: "Tom" | "Tomas" | "Tommy". После набора значений идет оператор
as, после которого указывается псевдоним. И вне зависимости от того, какая именно строка передана, она
окажется в переменной name.

Псевдоним можно применять как для отдельного значения шаблона, так и для всего шаблона:

```
def print_person(person):
    match person:
        case ("Tom" | "Tomas" as name, 37 | 38 as age):  # псевдонимы
для отдельных значений
            print(f"Tom| Name: {name}  Age: {age}")
        case ("Bob" | "Robert", 41 | 42) as bob:         # псевдоним
для всего шаблона
            print(f"Bob| Name: {bob[0]}  Age: {bob[1]}")
        case _:
            print("Undefined")


print_person(("Tomas", 38))    # Tom| Name: Tomas  Age: 38
print_person(("Robert", 41))   # Bob| Name: Robert  Age: 41
```

Обычно псевдонимы более применимы в каких-то более сложных по структуре данных. Например:

```
class Person:
    __match_args__ = ("name", "age")
    def __init__(self, name, age):
        self.name = name
        self.age = age


def print_family(family):
```

```
    match family:
        case (Person() as husband, Person() as wife):
            print(f"Husband. Name: {husband.name}  Age:
{husband.age}")
            print(f"Wife. Name: {wife.name}  Age: {wife.age}")
        case _:
            print("Undefined")


print_family((Person("Tom", 37), Person("Alice", 33)))
# Husband. Name: Tom  Age: 37
# Wife. Name: Alice  Age: 33

print_family((Person("Sam", 28), Person("Kate", 25)))
# Husband. Name: Sam  Age: 28
# Wife. Name: Kate  Age: 25
```

Здесь функция print_family принимает кортеж, который должен состоять из двух элементов Person. В первом шаблоне определяем
для первого элемента псевдоним husband, а для второго - псевдоним wife. Затем, используя эти псевдонимы, мы можкем обращаться к их атрибутам.