ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА КІБЕРНЕТИКИ КИЇВСЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ ІМЕНІ ТАРАСА ШЕВЧЕНКА

М.М. Верес, Л.О. Катеринич, О.О. Супрун

Програмування мовою Python

Навчальний посібник

Київ 2023

УДК 519.85 (075.8)

ББК 32.973.2-018я73

П11

Рецензенти:

д-р фіз.-мат. наук *С.С. Шкільняк,*

к-т. наук *Є.О.. Демківський.*

Рекомендовано до друку вченою радою факультету комп’ютерних наук та кібернетики (протокол №4 від “7” листопада 2023 року)

Верес М.М., Катеринич Л.О., Супрун О.О.

П11 Програмування мовою Python: навчальний посібник / Верес М.М., Катеринич Л.О., Супрун О.О.– Київ, 2023. – 362 с.

УДК 519.85 (075.8)

ББК 32.973.2-018я73

П11

В посібнику викладено основні принципи програмування та базову термінологію і навички програмування мовою Python. Складається з 2 частин основи програмування мовою Python та основи Web-програмування мовою Python. Текст містить велику кількість прикладів та завдання для самостійної роботи, що має сприяти кращому засвоєнню матеріалу.

Для студентів, які вивчають дисципліну " Програмування мовою Python ", а також навчаються за освітніми програмами, пов'язаними з інформаційними та комп'ютерними технологіями..

© М.М. Верес, Л.О. Катеринич, О.О. Супрун, 2023

ЗМІСТ

Вступ. 5
Частина 1. Основи програмування мовою Python . 6
Тема 1. Загальні особливості синтаксису. Базові оператори. Умовні оператори. Циклічні оператори. 6
Тема 2. Базові типи та структури даних. 24
Тема 3. Функції. Області видимості та простори імен. Аргументи функцій. 64
Тема 4. Ітератори. Генератори. Функції генератори. Вирази-генератори. 73
Тема 5. Об'єкти. Класи. Класові ієрархії. Абстрактні класи. 86
Тема 6. Обробка винятків. Модулі та пакети. Документування коду. 103
Тема 7. Процеси. Потоки. Підпрограми. Паралелізм. Конкурування. Синхронізація. 116
Тема 8. Графічні користувацькі інтерфейси: Бібліотека PyQt. 133
Частина 2. Основи Web-програмування мовою Python. 153
Тема 9. Встановлення та конфігурування фреймворку Django. Створення Django проектів. 153
Тема 10. Шаблони користувацьких інтерфейсів. Генерація та обробка форм. Валідація даних. 160
Тема 11. Контролери та обробка запитів користувачів. Робота з динамічними даними. Куки. Сесії. Фільтри. 185
Тема 12. Моделі та активні записи. Представлення, збереження та обробка даних. Міграції. Асоціації та відношення між типами. 211
Тема 13. Маршрутизація та обробка URL запитів. CRUD. REST. 226

Тема 14. Конфігурування система управління контентом. 243
Тема 15. Тестування та налагодження Django проектів. Фреймворк PyTest. 249

Практичні завдання 263
Умови лабораторних робіт 273
Варіанти завдань лабораторних робіт 274
Організація оцінювання 278
Лабораторна робота № 1 281
Лабораторна робота № 2 290
Лабораторна робота № 3 299
Лабораторна робота № 4 312
Лабораторна робота № 5 322
Лабораторна робота № 6 337
Лабораторна робота № 7 342
Лабораторна робота № 8 351

Рекомендована література. 359

ВСТУП

Python інтерпретована об'єктно-орієнтована мова програмування високого рівня зі строгою динамічною типізацією. Розроблена в 1990 році Гвідо ван Россумом. Структури даних високого рівня разом із динамічною семантикою та динамічним зв'язуванням роблять її привабливою для швидкої розробки програм, а також як засіб поєднування наявних компонентів. Python підтримує модулі та пакети модулів, що сприяє модульності та повторному використанню коду. Інтерпретатор Python та стандартні бібліотеки доступні як у скомпільованій, так і у вихідній формі на всіх основних платформах. В мові програмування Python підтримується кілька парадигм програмування, зокрема: об'єктно-орієнтована, процедурна, функціональна та аспектно-орієнтована.

Частина 1. Основи програмування мовою Python.

Тема 1. Загальні особливості синтаксису. Базові оператори. Умовні оператори. Циклічні оператори.

ОПЕРАТОРИ ПРИСВОЄННЯ

В мові програмування Python оператори присвоєння використовуються для присвоєння значень змінним. Основний оператор присвоєння в Python - це =. В Python також є деякі інші оператори присвоєння, які дозволяють виконувати певні операції одночасно з присвоєнням значення. Ось декілька прикладів:

Просте присвоєння: Основний оператор присвоєння в Python - це =. Він використовується для присвоєння значення змінній.

```
x = 10
# Змінній x присвоюється значення 10
```

Інші оператори присвоєння: Python також підтримує інші оператори присвоєння, такі як +=, -=, *=, /=, які використовуються для виконання операції та одночасного присвоєння.

```
x += 5 # Змінній x додається 5 (еквівалентно x = x + 5)
y *= 2 # Змінній y помножується на 2 (еквівалентно y =
y * 2)
```

Можливість присвоєння більше одного значення: Python дозволяє присвоювати більше одного значення в одному рядку.

```
a, b, c = 10, 20, 30
# Змінним a, b і c присвоюються значення 10, 20 і 30
відповідно
```

Ланцюжкове присвоєння: Можна присвоювати одну змінну іншій.

```
a = 5 b = a
# Змінній b присвоюється значення змінної a
```

Змінна може змінювати своє значення: Змінні можуть змінювати своє значення впродовж програми.

```
x = 10 x = 20
# Змінній x присвоюється нове значення 20
```

Багаторазове присвоєння: В одному рядку можна присвоїти однакове значення кільком змінним.

```
a = b = c = 0
# Змінним a, b і c присвоюється значення 0
```

Деструктивне присвоєння: Можна розпакувати значення з кортежу чи списку та присвоїти їх змінним.

```
кортеж = (10, 20, 30)
x, y, z = кортеж
# Змінним x, y і z присвоюються відповідні значення з
кортежу
```

Ці оператори присвоєння допомагають керувати змінними та їхніми значеннями в програмах на Python.

Присвоєння умовно: можна використовувати оператори присвоєння для умовних операцій. Наприклад, ви можете присвоїти значення змінній залежно від певної умови.

```
x = 10 if умова else 20
Змінній x присвоюється 10, якщо умова виконується, і
20, якщо не виконується.
```

Присвоєння за допомогою оператора *: В Python ви можете використовувати оператор * для розпакування значень зі списку або кортежу.

```
список = [1, 2, 3, 4, 5] перший, другий, *решта =
список
```

Змінним перший і другий присвоюються перші два елементи списку, а змінній решта - решта елементів у вигляді списку.

Присвоєння для ітерації: Ви можете використовувати цикл for для присвоєння значень зі списку змінним в ітераційному контексті.

```
числа = [1, 2, 3, 4, 5] for число in числа: # Операції
зі змінною "число"
```

Змінній число присвоюється кожне значення зі списку в кожній ітерації циклу.

АРИФМЕТИЧНІ ОПЕРАТОРИ

В Python існує низка арифметичних операторів, які використовуються для виконання арифметичних операцій над числами.

Основні арифметичні оператори включають:

Додавання (+): Використовується для додавання чисел.

```
a = 5
b = 3
c = a + b
# Результат: c = 8
```

Віднімання (-): Використовується для віднімання чисел.

```
a = 10
b = 7
c = a – b
# Результат: c = 3
```

Множення (*): Використовується для множення чисел.

```
a = 4
b = 6
c = a * b
# Результат: c = 24
```

Ділення (/): Використовується для ділення чисел. В результаті отримується десятковий результат, навіть якщо обидва операнди цілі числа.

```
a = 10
b = 3
c = a / b
# Результат: c = 3.3333333333333335
```

Цілочисельне ділення (//): Використовується для ділення чисел і повертає ціле число, округлене до найближчого меншого значення.

```
a = 10
b = 3
c = a // b
# Результат: c = 3
```

Модуль (%): Використовується для знаходження залишку від ділення двох чисел.

```
a = 10
b = 3
c = a % b
# Результат: c = 1
```

Піднесення в ступінь (**): Використовується для піднесення числа до певного ступеня.

```
a = 2
b = 3
c = a ** b
# Результат: c = 8
```

Ці арифметичні оператори використовуються для виконання різних математичних операцій у програмах на Python. Вони дозволяють вам працювати з числами і виконувати різні обчислення.

Наступні операції використовуються для виконання операцій і одночасного присвоєння результату змінним. Вони можуть бути корисні для оптимізації коду та його скорочення.

Оператор додавання і призначення (+=): Цей оператор використовується для додавання значення однієї змінної до іншої і призначення результату першій змінній. Він скорочує код для додавання.

```
a = 5
b = 3
a += b
# Результат: a = 8
```

Оператор віднімання і призначення (-=): Аналогічно до попереднього, але для віднімання.

```
a = 10
b = 7
a -= b
# Результат: a = 3
```

Оператор множення і призначення (*=): Аналогічно для множення.

```
a = 4
b = 6
a *= b
# Результат: a = 24
```

Оператор ділення і призначення (/=): Аналогічно для ділення.

```
a = 10
b = 3
a /= b
# Результат: a = 3.3333333333333335
```

Оператор цілочисельного ділення і призначення (//=): Аналогічно для цілочисельного ділення.

```
a = 10
b = 3
a //= b
# Результат: a = 3
```

Оператор модуля і призначення (%=): Аналогічно для обчислення залишку.

```
a = 10
```

```
b = 3
a %= b
# Результат: a = 1
```

Оператор піднесення в ступінь і призначення (=):** Використовується для піднесення числа до певного ступеня і одночасного присвоєння результату змінній.

```
a = 2
b = 3
a **= b
# Результат: a = 8
```

Ці оператори присвоєння роблять код більш зрозумілим і коротшим, оскільки вони виконують операцію і присвоєння значення одночасно. Вони допомагають уникнути додаткових операцій присвоєння та роблять код більш ефективним.

ОПЕРАТОРИ ПОРІВНЯННЯ

В мові програмування Python існує низка операторів порівняння, які використовуються для порівняння значень і повернення результату у вигляді логічного значення (True або False). Основні оператори порівняння включають:

Рівність (==): Порівнює два значення на рівність.

```
a = 5
b = 5
результат = (a == b)
# Результат: результат = True
```

Нерівність (!=): Перевіряє, чи два значення не рівні один одному.

```
a = 5
b = 3
результат = (a != b)
# Результат: результат = True
```

Більше (>): Перевіряє, чи перше значення більше другого.

```
a = 7
b = 4
результат = (a > b)
# Результат: результат = True
```

Менше (<): Перевіряє, чи перше значення менше другого.

```
a = 3
b = 6
результат = (a < b)
# Результат: результат = True
```

Більше або рівне (>=): Перевіряє, чи перше значення більше або рівне другому.

```
a = 5
b = 5
результат = (a >= b)
# Результат: результат = True
```

Менше або рівне (<=): Перевіряє, чи перше значення менше або рівне другому.

```
a = 4
b = 5
результат = (a <= b)
# Результат: результат = True
```

Оператор ідентичності (is): Оператор is використовується для порівняння ідентичності об'єктів. Він перевіряє, чи дві змінні посилаються на один і той самий об'єкт в пам'яті.

```
a = [1, 2, 3]
b = a
# b посилається на той самий об'єкт, що і a результат =
(a is b)
# Результат: результат = True
```

Оператор відмінності (is not): Оператор is not перевіряє, чи дві змінні посилаються на різні об'єкти.

```
a = [1, 2, 3]
b = [1, 2, 3]
```

```
# b посилається на інший об'єкт, ніж a результат = (a
is not b)
# Результат: результат = True
```

Оператор членства (in): Оператор in використовується для перевірки, чи елемент знаходиться в об'єкті, як от рядку, списку, кортежі тощо.

```
текст = "Привіт, світ"
результат = ("світ" in текст)
# Результат: результат = True
```

Оператор невходження (not in): Оператор not in використовується для перевірки, чи елемент не знаходиться в об'єкті.

```
числа = [1, 2, 3, 4, 5]
результат = (6 not in числа)
# Результат: результат = True
```

Оператор порівняння рядків: Рядки можна порівнювати за лексикографічним порядком, де більший рядок має більший значення. Оператори порівняння працюють із рядками:

```
рядок1 = "abc"
рядок2 = "def"
результат = (рядок1 < рядок2)
# Результат: результат = True, оскільки "abc" менше за
"def" за лексикографічним порядком
```

Оператор порівняння списків і інших ітерабельних об'єктів: Ви можете порівнювати списки та інші ітерабельні об'єкти за допомогою операторів порівняння, які порівнюють їхні елементи.

```
список1 = [1, 2, 3]
список2 = [1, 2, 3]
результат = (список1 == список2)
# Результат: результат = True, оскільки обидва списки
містять однакові елементи
```

Оператор порівняння кортежів і інших ітерабельних об'єктів: Порівнювати можна не лише списки, а й інші ітерабельні об'єкти, такі як кортежі.

```
кортеж1 = (1, 2, 3)
кортеж2 = (4, 5, 6)
результат = (кортеж1 != кортеж2)
# Результат: результат = True, оскільки кортежі різні
```

Оператори порівняння допомагають порівнювати різні типи даних, включаючи числа, рядки, списки, кортежі та інші об'єкти, і визначати, чи вони рівні або різні. Вони є важливою частиною умовних виразів та операцій порівняння в Python.

ПОБІТОВІ ОПЕРАТОРИ

У мові програмування Python є низка побітових операторів, які виконують операції над бітами (найменшими одиницями інформації в бінарному коді). Побітові оператори використовуються для маніпуляції бітами в цілочисельних значеннях.

Основні побітові оператори включають:

Побітовий AND (&): Порівнює біти двох чисел та повертає результат, де біти встановлені на 1, якщо вони встановлені на 1 в обох числах.

```
a = 5
# 0b0101 в бінарній формі
b = 3
# 0b0011 в бінарній формі
результат = a & b
# Результат: результат = 1 (0b0001 в бінарній формі)
```

Побітовий OR (|): Порівнює біти двох чисел та повертає результат, де біти встановлені на 1, якщо вони встановлені на 1 в одному або обох числах.

```
a = 5
# 0b0101 в бінарній формі
b = 3 # 0b0011 в бінарній формі
```

```
результат = a | b
# Результат: результат = 7 (0b0111 в бінарній формі)
```

Побітовий XOR (^): Порівнює біти двох чисел та повертає результат, де біти встановлені на 1, якщо вони встановлені на 1 в одному, але не в обох числах.

```
a = 5
# 0b0101 в бінарній формі
b = 3
# 0b0011 в бінарній формі
результат = a ^ b
# Результат: результат = 6 (0b0110 в бінарній формі)
```

Побітовий NOT (~): Інвертує всі біти числа, змінюючи 0 на 1 і 1 на 0.

```
a = 5
# 0b0101 в бінарній формі
результат = ~a
# Результат: результат = -6
```

Побітовий зсув вліво (<<): Зсуває біти числа вліво на задану кількість позицій.

```
a = 5
# 0b0101 в бінарній формі
результат = a << 2
# Результат: результат = 20 (0b10100 в бінарній формі)
```

Побітовий зсув вправо (>>): Зсуває біти числа вправо на задану кількість позицій.

```
a = 20
# 0b10100 в бінарній формі
результат = a >> 2
# Результат: результат = 5 (0b0101 в бінарній формі)
```

Зсув з заповненням нулями вліво (<<) з оператором призначення (<<=): Цей оператор виконує зсув бітів числа вліво і заповнює порожні позиції нулями.

```
a = 5
# 0b0101 в бінарній формі
a <<= 2
```

```
# Виконує зсув вліво на 2 позиції з заповненням нулями
# Результат: a = 20 (0b10100 в бінарній формі)
```

Зсув з заповненням нулями вправо (>>) з оператором призначення (>>=): Цей оператор виконує зсув бітів числа вправо і заповнює порожні позиції нулями.

```
a = 20
# 0b10100 в бінарній формі
a >>= 2
# Виконує зсув вправо на 2 позиції з заповненням нулями
# Результат: a = 5 (0b0101 в бінарній формі)
```

Побітовий AND з оператором призначення (&=): Цей оператор виконує побітовий AND над бітами чисел і призначає результат першому числу.

```
a = 5
# 0b0101 в бінарній формі
b = 3
# 0b0011 в бінарній формі
a &= b
# Виконує побітовий AND і призначає результат змінній
"a"
# Результат: a = 1 (0b0001 в бінарній формі)
```

Побітовий OR з оператором призначення (|=): Цей оператор виконує побітовий OR над бітами чисел і призначає результат першому числу.

```
a = 5
# 0b0101 в бінарній формі
b = 3
# 0b0011 в бінарній формі
a |= b
# Виконує побітовий OR і призначає результат змінній
"a"
 # Результат: a = 7 (0b0111 в бінарній формі)
```

Ці розширені побітові оператори дозволяють виконувати побітові операції та призначати результати одночасно, що може спростити код і підвищити його читабельність в деяких ситуаціях.

Побітовий XOR з оператором призначення (^=): Цей оператор виконує побітовий XOR над бітами чисел і призначає результат першому числу.

```
a = 5
# 0b0101 в бінарній формі
b = 3
# 0b0011 в бінарній формі
a ^= b
# Виконує побітовий XOR і призначає результат змінній
"a"
# Результат: a = 6 (0b0110 в бінарній формі)
```

Побітові оператори і їхні розширені версії дозволяють виконувати операції над бітами чисел і маніпулювати ними на рівні бінарного подання даних. Вони зазвичай використовуються для роботи з масками, операціями над флагами, оптимізацією обчислень та іншими подібними завданнями.

ЛОГІЧНІ ОПЕРАТОРИ

Логічні оператори в мові програмування Python використовуються для виконання логічних операцій над булевими значеннями (True або False). Основні логічні оператори включають:

Логічне І (and): Цей оператор повертає True, якщо обидва операнди є True.

```
x = True
y = False
результат = x and y
# Результат: результат = False
```

Логічне АБО (or): Цей оператор повертає True, якщо хоча б один із операндів є True.

```
x = True
y = False
результат = x or y
```

```
# Результат: результат = True
```

Логічне НЕ (not): Цей оператор інвертує булеве значення, тобто True перетворюється на False, і навпаки.

```
x = True
результат = not x
# Результат: результат = False
```

Логічні оператори використовуються для створення складних умовних виразів та контролю потоку програми. Вони дозволяють комбінувати і перевіряти умови для прийняття рішень у програмах.

ОПЕРАТОРИ НАЛЕЖНОСТІ

Оператори належності в мові програмування Python використовуються для перевірки належності елемента до контейнера, такого як список, кортеж, рядок тощо.

Основні оператори належності включають:

Оператор in: Цей оператор перевіряє, чи елемент належить контейнеру, і повертає булеве значення True, якщо належить, та False, якщо не належить.

```
текст = "Привіт, світ"
результат = "світ" in текст
# Результат: результат = True
```

Оператор not in: Цей оператор перевіряє, чи елемент не належить контейнеру, і повертає булеве значення True, якщо не належить, та False, якщо належить.

```
числа = [1, 2, 3, 4, 5]
результат = 6 not in числа
# Результат: результат = True
```

Ці оператори належності дуже корисні для перевірки наявності елементів в контейнерах та прийняття рішень в програмах на основі наявності або відсутності певних значень. Вони дозволяють легко визначати, чи

конкретний елемент знаходиться у списку, кортежі, рядку або іншому контейнері.

ОПЕРАТОРИ ТОТОЖНОСТІ

В мові програмування Python існують оператори тотожності, які дозволяють порівнювати об'єкти за їхньою ідентичністю, тобто чи це один і той самий об'єкт в пам'яті. Основні оператори тотожності включають:

Оператор is: Цей оператор перевіряє, чи дві змінні посилаються на один і той самий об'єкт. Він повертає True, якщо об'єкти ідентичні, і False, якщо вони різні.

```
список1 = [1, 2, 3]
список2 = список1
# список2 посилається на той самий об'єкт, що і список1
результат = список1 is список2
# Результат: результат = True
```

Оператор is not: Цей оператор перевіряє, чи дві змінні не посилаються на один і той самий об'єкт. Він повертає True, якщо об'єкти різні, і False, якщо вони ідентичні.

```
список1 = [1, 2, 3]
список2 = [1, 2, 3]
# список2 посилається на інший об'єкт, ніж список1
результат = список1 is not список2
# Результат: результат = True
```

Оператори тотожності корисні для порівняння об'єктів на рівні пам'яті. Вони допомагають визначити, чи дві змінні посилаються на один і той самий об'єкт або на різні об'єкти.

ПРІОРИТЕТНІСТЬ ОПЕРАТОРІВ

Базові оператори мови Python мають наступну пріоритетність:

- піднесення до степеня **;
- побітове доповнення ~, унарний + та унарний -;
- множення *, ділення /, ділення по модулю %,
- цілочисельне ділення //;
- додавання +, віднімання -;
- побітовий зсув ліворуч <<, побітовий зсув праворуч >>;
- побітова кон'юнкція &;
- побітова диз'юнкція |,побітова виключаюча диз'юнкція ^;
- оператори порівняння <, >, <=, >=, !=, ==;
- оператори присвоєння =, %=, /=, //=, -=, +=, *=, **=, >>=, <<=, &=, ^=, |=;
- оператори тотожності is, is not;
- оператори належності in, not in;
- логічні оператори and, or, not.

УМОВНІ ОПЕРАТОРИ МОВИ PYTHON

ОПЕРАТОР IF

Оператор if в Python - це один із основних елементів умовних виразів, який використовується для виконання певних блоків коду в залежності від умов. Він дозволяє програмі приймати рішення на основі значень змінних або виразів. Оператор if виконує блок коду, якщо умова, вказана всередині нього, є істинною (True), і може бути доповнений іншими операторами для обробки альтернативних сценаріїв.

```
if умова:
      # Блок коду, який виконується, якщо умова істинна
```

Приклад:

```
x = 10
if x > 5:
     print("x більше 5")
```

ОПЕРАТОР IF-ELSE

Оператор if-else в Python - це конструкція, яка дозволяє виконувати один блок коду, якщо певна умова істинна (True), і інший блок коду, якщо умова є хибною (False). Ця конструкція часто використовується для вибору між двома альтернативними діями в залежності від виконання умови.

```
if умова:
     # Блок коду, який виконується, якщо умова істинна
else:
     # Блок коду, який виконується, якщо умова хибна
```

Приклад:

```
x = 10
if x > 5:
    print("x більше 5")
else:
    print("x не більше 5")
```

Оператор if-else може бути корисним для обробки ситуацій, коли потрібно визначити альтернативну дію в разі невиконання умови.

ТЕРНАРНА ФОРМА

У Python існує тернарна форма оператора if, яка дозволяє коротко визначити значення змінної на основі умови. Тернарний оператор має наступний синтаксис:

```
вираз_if_true if умова else вираз_if_false
```

Цей оператор оцінює умову, і якщо умова істинна (True), він повертає перший вираз (значення), а в іншому випадку він повертає другий вираз (значення).
Приклад:

```
x = 10
result = "x більше 5" if x > 5 else "x не більше 5"
print(result)
```

Тернарна форма оператора if корисна для коротких умовних виразів, де потрібно обрати одне значення або інше, в залежності від умови, і вона полегшує читання і розуміння коду.

ОПЕРАТОР IF-ELIF-ELSE

Оператор if-elif-else в Python - це конструкція, яка дозволяє вибирати між декількома альтернативними блоками коду в залежності від різних умов. Використовується, коли є потреба обробити кілька варіантів вхідних даних або різні умови.

```
if умова1:
     # Блок коду, який виконується, якщо умова1 істинна
elif умова2:
     # Блок коду, який виконується, якщо умова2 істинна
elif умова3:
     # Блок коду, який виконується, якщо умова3 істинна
else:
      # Блок коду, який виконується, якщо жодна з умов
      не є істинною
```

Послідовність умов і блоків коду означає, що спочатку перевіряється умова1. Якщо вона істинна, виконується відповідний блок коду, і виконання оператора завершується. Якщо умова1 не є істинною, перевіряється умова2, і так далі, поки не буде знайдено першу істинну умову або не спрацює блок коду в розділі else.

```
x = 10
if x > 15:
    print("x більше 15")
elif x > 5:
    print("x більше 5, але не більше 15")
else:
    print("x не більше 5")
```

ОСОБЛИВОСТІ ОБЧИСЛЕННЯ ІСТИННОСТІ

Перевірка логічної істинності виразів у мові Python має ряд наступних особливостей:

- будь-яке число відмінне від нуля або непорожній об'єкт інтерпретується як істинна, тобто True;
- числа, що рівні нулю, порожні об'єкти та спеціальний об'єкт None інтерпретуються як хиба, тобто False;
- операції порівняння та перевірки на рівність застосовуються до структур даних рекурсивно.

ТЕМА 2. БАЗОВІ ТИПИ ТА СТРУКТУРИ ДАНИХ.

Python підтримує різні типи даних для роботи з числовою інформацією. Основні числові типи у Python включають:

- int (цілі числа): Цей тип даних використовується для цілих чисел, як позитивних, так і негативних, а також нуля. Наприклад: `x = 5 y = -10`
- float (числа з плаваючою комою): Цей тип даних використовується для чисел з плаваючою комою, таких як дійсні числа. Наприклад: `a = 3.14 b = -0.5`
- complex (комплексні числа): Цей тип даних використовується для комплексних чисел з виглядом a + bj, де a і b - це дійсні числа, а j - уявна одиниця. Наприклад: `z = 2 + 3j`
- bool (логічні значення): Цей тип даних має два можливих значення - True (Правда) і False (Неправда). Використовується для логічних операцій. Наприклад: `is_true = True is_false = False`
- Decimal (десяткові числа): Цей тип даних використовується для виконання обчислень з великою точністю на основі десяткової системи числення, у відміну від чисел з плаваючою комою. Щоб використовувати його, потрібно імпортувати модуль decimal. Наприклад:

```
from decimal import Decimal
dec = Decimal('10.5')
```

- Fraction (дроби): Цей тип даних використовується для роботи з раціональними дробами. Для цього потрібно імпортувати модуль fractions. Наприклад:

```
from fractions import Fraction
frac = Fraction(3, 4)
```

Ці числові типи даних дозволяють виконувати різноманітні операції та обчислення у Python в залежності від потреб вашої програми.

У Python можна застосовувати різні математичні операції та функції до чисел. Ось кілька прикладів операцій та функцій, які можна використовувати для чисел:

- Арифметичні операції:
    - Додавання +
    - Віднімання -
    - Множення *
    - Ділення /
    - Цілочисельне ділення //
    - Остача від ділення %
    - Піднесення до степеня **

Наприклад:

```
a = 10
b = 3
sum_result = a + b
difference_result = a – b
product_result = a * b
division_result = a / b
```

- Функції для роботи з числами:
    - abs(x): Повертає абсолютне значення числа x.
    - round(x, n): Округлює число x до n знаків після коми.
    - max(iterable): Повертає найбільший елемент у послідовності.
    - min(iterable): Повертає найменший елемент у послідовності.
    - sum(iterable): Сумує всі елементи у послідовності.

Наприклад:

```
absolute_value = abs(-5)
rounded_value = round(3.14159, 2)
maximum = max([4, 8, 2, 10])
total = sum([1, 2, 3, 4, 5])
```

- Математичні функції:
    - math.sqrt(x): Обчислює квадратний корінь числа x (потрібно імпортувати модуль math).
    - math.sin(x), math.cos(x), math.tan(x): Обчислюють тригонометричні функції (потрібно імпортувати модуль math).

Наприклад:

```
import math
sqrt_result = math.sqrt(25)
sine_result = math.sin(math.pi/2)
```

Це лише деякі з операцій та функцій, які можна використовувати для чисел у Python. Python надає багато можливостей для роботи з числами і математикою взагалі.

РЯДКИ

Рядки (strings) в Python представляють собою послідовність символів і є одним з основних типів даних. Вони можуть бути подвійними (") або одинарними (') лапками, і можуть містити будь-які символи, включаючи букви, цифри, символи пунктуації та пробіли.

Приклади рядків у Python:

Створення рядка:

```
my_string = "Привіт, світ!"
# Рядок в подвійних лапках
another_string = 'Це також рядок'
# Рядок в одинарних лапках
```

Конкатенація рядків (з'єднання рядків):

```
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
print(full_name)
# Виведе "John Doe"
```

Доступ до символів у рядку за індексом:

```
my_string = "Hello, World!"
first_char = my_string[0]
# Перший символ (H)
second_char = my_string[7]
# Восьмий символ (W)
```

Довжина рядка (кількість символів):

```
my_string = "Python"
length = len(my_string) # Довжина рядка (6)
```

Зрізи рядків (slicing):

```
my_string = "Hello, World!"
substring = my_string[0:5]
# Зріз від 0 до 5 (включно), "Hello"
```

Рядкові методи: Python має багато вбудованих методів для роботи з рядками, такі як split(), strip(), upper(), lower(), replace(), і багато інших. Наприклад:

```
sentence = "Це приклад речення."
words = sentence.split()
# Розділити рядок на слова
uppercase = sentence.upper()
# Перетворити на великі літери
```

Перевірка наявності підрядка в рядку:

```
my_string = "Python is great"
contains = "is" in my_string
# Перевірити, чи міститься "is" у рядку (True)
```

Форматування рядків: Python надає різні способи форматування рядків, включаючи використання f-строк, методу format() та оператора %. Наприклад:

```
name = "Alice"
age = 30
formatted_string = f"Мене звуть {name} і мені {age}
років."
```

Рядки в Python є невід'ємною частиною програмування і використовуються для зберігання, обробки та виведення текстової інформації.

ОПЕРАЦІЇ НАД РЯДКАМИ.

У Python існує багато операцій та методів для роботи з рядками. Ось деякі з найпоширеніших операцій над рядками:

Конкатенація рядків: Для об'єднання (з'єднання) рядків ви можете використовувати оператор + або метод join(). Наприклад:

```
str1 = "Hello"
str2 = "World"
result = str1 + ", " + str2
# "Hello, World"
```

Повторення рядків: Ви можете повторювати рядок за допомогою оператора *. Наприклад:

```
word = "Python" repeated = word * 3
 # "PythonPythonPython"
```

Довжина рядка: Функція len() визначає довжину рядка, тобто кількість символів у ньому. Наприклад:

```
     text = "Hello, World!" length = len(text) # 13
```

Зрізи рядків (slicing): Ви можете витягти підрядок з рядка, використовуючи зрізи. Наприклад:

```
     text = "Python Programming" substring = text[7:18]
# "Programming"
```

Перетворення регістру: Методи upper() та lower() використовуються для перетворення рядка на великі або малі літери відповідно. Наприклад:

```
text = "Hello, World!"
upper_text = text.upper()
# "HELLO, WORLD!"
lower_text = text.lower()
# "hello, world!"
```

Видалення пропусків: Методи strip(), lstrip() і rstrip() використовуються для видалення пропусків з початку та кінця рядка. Наприклад:

```
text = " Пробіли "
trimmed_text = text.strip()
# "Пробіли"
```

Заміна підрядка: Метод replace() використовується для заміни підрядка на інший підрядок. Наприклад:

```
sentence = "Python is fun"
new_sentence = sentence.replace("Python","Programming")
# "Programming is fun"
```

Перевірка підрядка: Ви можете перевірити, чи міститься певний підрядок у рядку, використовуючи оператор in. Наприклад:

```
text = "Python is great"
contains = "is" in text
# True
```

Розділення рядка: Метод split() дозволяє розділити рядок на список підрядків на основі певного розділника (за замовчуванням - пробіли). Наприклад:

```
sentence = "Python is an interpreted language"
words = sentence.split()
# ["Python", "is", "an", "interpreted", "language"]
```

Порівняння рядків: Порівняння рядків можна виконувати за допомогою операторів порівняння (==, !=, <, >, <=, >=). Наприклад:

```
str1 = "apple"
str2 = "banana"
result = str1 < str2
# True (порівняння за лексикографічним порядком)
```

Перевірка на початок або кінець рядка: Методи startswith() та endswith() дозволяють перевірити, чи рядок починається або закінчується певним підрядком. Наприклад:

```
text = "Hello, World!"
starts_with_hello = text.startswith("Hello")
# True ends_with_exclamation = text.endswith("!")
# True
```

Форматування рядків за допомогою методу format(): Ви можете вставляти значення в рядок, використовуючи метод format(). Наприклад:

```
name = "Alice"
age = 30
formatted_string = "Мене звуть {} і мені {}
років.".format(name, age)
```

Форматування рядків за допомогою f-строк: F-строки (f-strings) - це спосіб форматування рядків, який дозволяє вставляти значення прямо в рядок. Наприклад:

```
name = "Bob"
age = 25
formatted_string = f"Мене звуть {name} і мені {age}
років."
```

Видалення символів з рядка за певною умовою: Методи strip(), lstrip(), та rstrip() можна використовувати для видалення символів з початку та кінця рядка, задовольняючи певну умову. Наприклад:

```
text = "----Привіт----"\
cleaned_text = text.strip('-')
# "Привіт"
```

Форматування чисел у рядку: Ви можете використовувати методи форматування для чисел, такі як

str.format() для задання точності, кількості знаків після коми тощо.

```
value = 3.14159265359
formatted_value = "{:.2f}".format(value)
# "3.14"
```

Переведення інших типів даних у рядок: використовується функцію str() для перетворення інших типів даних (наприклад, чисел або списків) у рядок. Наприклад:

```
num = 42
num_str = str(num)
# "42"
```

Форматування рядка за допомогою оператора %: Даний оператор може бути використаний для форматування рядка аналогічно методу format().

```
name = "Charlie"
age = 28
formatted_string = "Мене звуть %s і мені %d років." %
(name, age)
```

Порівняння регістрів: Методи isupper(), islower() та istitle() використовуються для перевірки регістру рядка. Наприклад:

```
text = "Hello, World!"
is_upper = text.isupper()
# False is_lower = text.islower()
# False is_title = text.istitle()
# True
```

Пошук індексу підрядка: Метод find() шукає підрядок у рядку і повертає перший індекс його знаходження. Якщо підрядок не знайдено, повертається -1. Наприклад:

```
text = "Python is fun"
index = text.find("is")
# 7
```

Перевертання рядка: Ви можете використовувати зрізи, щоб перевернути рядок. Наприклад:

```
text = "Python"
reversed_text = text[::-1]
# "nohtyP"
```

Зміна регістру символів: Методи capitalize(), title(), swapcase() використовуються для зміни регістру символів в рядку. Наприклад:

```
text = "python programming is fun"
capitalized_text = text.capitalize()
# "Python programming is fun"
title_text = text.title()
# "Python Programming Is Fun"
swapped_text = text.swapcase()
# "PYTHON PROGRAMMING IS FUN"
```

Перевірка символів у рядку: Методи isalpha(), isdigit(), isalnum() та інші дозволяють перевіряти, чи відповідають символи певним умовам. Наприклад:

```
text1 = "Hello"
text2 = "123"
is_alpha = text1.isalpha()
# True is_digit = text2.isdigit()
# True
```

Злиття рядків зі списків: Ви можете злити рядки зі списків за допомогою методу join(). Наприклад:

```
words = ["Python", "is", "fun"]
sentence = " ".join(words)
# "Python is fun"
```

Розбиття рядка на рядкові літерали: Python підтримує рядкові літерали, які можна використовувати для запису багаторядкових рядків. Наприклад:

```
multi_line_text = """Це рядковий літерал, що містить
кілька рядків тексту."""
```

МОДУЛЬ STRING

Модуль string в Python - це вбудований модуль, який містить константи та функції, пов'язані з роботою зі стрічками (рядками). Цей модуль надає корисні ресурси для роботи з рядками, такі як константи для легкого доступу до різних символьних класів та функції для перетворення регістрів і знаків пунктуації. Ось деякі з важливих частин модуля string:

Константи символьних класів: Модуль string містить різні константи, які представляють класи символів. Наприклад, string.ascii_letters містить всі літери (букви) обох регістрів, а string.digits містить цифри.

```
import string
all_letters = string.ascii_letters
#abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
digits = string.digits
# "0123456789"
```

Методи для роботи з регістром: Модуль string надає корисні методи для перетворення регістрів символів. Наприклад, string.uppercase містить всі великі літери, і ви можете використовувати метод str.upper() для перетворення рядка у верхній регістр.

```
import string text = "Hello, World!"
uppercase_text = text.upper()
# "HELLO, WORLD!"
```

Константи для знаків пунктуації: Модуль містить константи для різних знаків пунктуації, такі як string.punctuation. Вони можуть бути корисні для обробки рядків та видалення зайвих символів.

```
import string
punctuation_chars = string.punctuation
# "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~"
```

Корисні функції: Модуль містить деякі корисні функції, наприклад, string.capwords(), яка великими літерами перші букви слів у рядку.

```
import string
text = "python is fun"
capitalized_text = string.capwords(text)
# "Python Is Fun"
```

Модуль string надає зручні засоби для обробки рядків, особливо коли вам потрібно працювати з символами, регістрами, класами символів тощо. Використання цього модуля може полегшити роботу з рядками в вашому коді.

МОДУЛЬ TEXTWRAP

Модуль textwrap в Python є частиною стандартної бібліотеки і надає зручні засоби для роботи з текстовими рядками та форматування тексту. Цей модуль дозволяє виконувати такі завдання, як розрив рядків, вирівнювання тексту, заповнення абзаців, а також інші операції, пов'язані з форматуванням тексту.

Основними функціями модуля textwrap є: `textwrap.wrap(text, width, **kwargs)`: Ця функція розриває текст text на рядки так, щоб жоден рядок не був довшим за задану ширину width. Вона повертає список рядків, які представляють розриваний текст.

```
import textwrap
text = "This is a long text that needs to be wrapped to
fit within a certain width."
wrapped_text = textwrap.wrap(text, width=20) for line
in wrapped_text: print(line)
```

`textwrap.fill(text, width, **kwargs)`: Ця функція робить схожу операцію, але повертає одну рядок, де рядки розділені символом нового рядка для форматування тексту.

```
import textwrap
```

```
text = "This is a long text that needs to be wrapped to fit within a certain width."
formatted_text = textwrap.fill(text, width=20)
print(formatted_text)
```

Інші корисні функції: Модуль textwrap містить інші корисні функції, такі як `textwrap.shorten()` для скорочення тексту, `textwrap.indent()` для відступу тексту, `textwrap.dedent()` для зняття відступу тощо.

Опції та параметри: Функції модуля textwrap приймають деякі додаткові параметри, які дозволяють налаштувати форматування тексту, такі як subsequent_indent, expand_tabs, replace_whitespace, drop_whitespace, initial_indent, тощо.

Цей модуль корисний для виводу тексту в консолі, форматування довгих абзаців, створення зручного виводу тексту в файлі або для роботи з текстом у графічному інтерфейсі користувача. Він надає засоби для зручного керування форматуванням тексту та полегшує роботу з текстовими даними.

МОДУЛЬ RANDOM

Модуль random в Python є частиною стандартної бібліотеки і надає можливості для генерації випадкових чисел, вибору випадкових об'єктів і створення випадкових послідовностей. Цей модуль дозволяє створювати псевдовипадкові дані, які можуть бути корисні в багатьох сценаріях, таких як імітація, генерація тестових даних, розсіювання точок, випробовування алгоритмів тощо.

Основні можливості модуля random включають:

Генерація випадкових цілих чисел:

`random.randint(a, b)`: Генерує випадкове ціле число в інтервалі від a до b включно.

Генерація випадкових дійсних чисел:

`random.random()`: Генерує випадкове дійсне число в інтервалі [0.0, 1.0).

`random.uniform(a, b)`: Генерує випадкове дійсне число в інтервалі від a до b.

Генерація випадкових послідовностей:

`random.choice(seq)`: Вибирає випадковий елемент із послідовності seq.

`random.sample(seq, k)`: Вибирає k випадкових елементів із послідовності seq без повторень.

`random.shuffle(seq)`: Випадковим чином перемішує елементи в послідовності seq.

Генерація випадкових рядків:

Ви можете використовувати модуль random для генерації випадкових рядків, обираючи випадкові символи зі списку можливих символів.

Засоби встановлення "сідла" генератора випадкових чисел:

`random.seed(x)`: Встановлює "сідло" генератора випадкових чисел, що дозволяє відтворювати випадкові послідовності, якщо використовується те саме "сідло".

Робота з генератором випадкових чисел: Модуль random також дозволяє створити об'єкт генератора випадкових чисел за допомогою random.Random(). Це корисно, коли вам потрібно мати кілька генераторів і керувати ними окремо.

Нижче наведений приклад використання модуля random для генерації випадкового цілого числа та вибору випадкового елемента зі списку:

```
import random
# Генерація випадкового цілого числа в інтервалі [1, 6]
random_integer = random.randint(1, 6)
# Вибір випадкового елемента зі списку fruits =
["apple", "banana", "cherry", "date", "elderberry"]
random_fruit = random.choice(fruits)
print("Випадкове ціле число:", random_integer)
print("Випадковий фрукт:", random_fruit)
```

Модуль random дозволяє вам здійснювати генерацію випадкових значень для різноманітних завдань та досліджень, де потрібна випадковість.

СПИСКИ

Списки в Python - це один із важливих типів даних, який дозволяє зберігати колекцію об'єктів у впорядкованому порядку. Списки є змінюваними, тобто ви можете додавати, видаляти та змінювати їхні елементи. Ось деякі основні операції над списками в Python:

Створення списку: Ви можете створити список, перераховуючи елементи у квадратних дужках і розділяючи їх комами.

```
my_list = [1, 2, 3, 4, 5]
```

Додавання елементів:
`append()`: Додає елемент в кінець списку.

```
my_list.append(6)
```

Видалення елементів:
`remove()`: Видаляє перший знайдений елемент зі списку за вмістом.

```
my_list.remove(3)
```

Доступ до елементів: Ви можете отримати доступ до елементів списку за допомогою індексів (індексування), де індекси починаються з 0.

```
first_element = my_list[0]
# Перший елемент
last_element = my_list[-1]
# Останній елемент
```

Зрізи (Slicing): Ви можете отримати підсписок списку за допомогою зрізів.

```
sub_list = my_list[1:4]
# Елементи з індексами 1, 2, 3
```

Довжина списку: Можна визначити кількість елементів у списку за допомогою функції `len()`.

```
length = len(my_list)
# Кількість елементів у списку
```

Сортування:
`sort()`: Сортує список в порядку зростання.
`reverse()`: Розвертає порядок елементів у списку.

```
my_list.sort()
# Сортування в порядку зростання
my_list.reverse()
# Розвернути список
```

Перевірка наявності елемента: Ви можете перевірити, чи міститься певний елемент у списку за допомогою оператора in.

```
is_present = 3 in my_list
# Перевірка, чи міститься число 3 у списку
```

З'єднання списків:
+: З'єднує два списки.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined_list = list1 + list2
# [1, 2, 3, 4, 5, 6]
```

Копіювання списку:
`copy()`: Створює копію списку.

```
original_list = [1, 2, 3]
copied_list = original_list.copy()
```

Заповнення списку:
`extend()`: Додає елементи із іншого списку в поточний список.

```
my_list = [1, 2, 3]
my_list.extend([4, 5, 6])
# [1, 2, 3, 4, 5, 6]
```

Очищення списку:
`clear()`: Видаляє всі елементи із списку, залишаючи його порожнім.

```
my_list.clear()
# Порожній список
```

Видалення елементу за індексом:
`pop()`: Видаляє та повертає елемент за вказаним індексом (за замовчуванням - останній елемент).

```
my_list = [1, 2, 3, 4, 5]
removed_element = my_list.pop(2)
# Видаляє 3 та повертає його
```

Видалення першого елементу з певним значенням:
`remove()`: Видаляє перший елемент із списку, який має вказане значення.

```
my_list = [1, 2, 3, 4, 3, 5]
my_list.remove(3)
# Видаляє перший елемент зі значенням 3
```

Вставка елементу за індексом:
`insert()`: Вставляє елемент на певне місце у списку за вказаним індексом.

```
my_list = [1, 2, 3, 5]
my_list.insert(2, 4)
# Вставляє число 4 після 3
```

Зміна значення елементу за індексом:.

```
my_list = [1, 2, 3, 4, 5]
my_list[2] = 6
# Змінює значення елементу з індексом 2 на 6
```

Перевірка списку на порожнечу:
`len()`: Визначає кількість елементів у списку, і ви можете перевірити, чи він порожній за допомогою len().

```
my_list = []
is_empty = len(my_list) == 0
# True, список порожній
```

Копіювання списку (клонування):

```
original_list = [1, 2, 3]
copied_list = original_list.copy()
# Клонування списку
```

Знаходження індексу елемента за значенням:
`index()`: Знаходить індекс першого елемента з вказаним значенням.

```
my_list = [10, 20, 30, 40, 50]
index = my_list.index(30)
# Знаходить індекс елемента зі значенням 30
```

`Змішування списку (shuffle):` Модуль random може використовуватися для випадкового перемішування списку, змінюючи порядок його елементів.

```
import random
my_list = [1, 2, 3, 4, 5]
random.shuffle(my_list)
# Випадкове перемішування списку
```

Розширені операції над списками: Python також надає багато інших функцій та методів для роботи зі списками, такі як count(), sort(), reverse(), clear(), copy(), тощо.

Списки надають велику гнучкість для зберігання та обробки даних, і вони використовуються в багатьох програмах для організації даних.

Сума елементів у списку:
`sum()`: Обчислює суму всіх елементів у списку, якщо всі елементи є числами.

```
my_list = [1, 2, 3, 4, 5]
total = sum(my_list)
# 15
```

Зміна розміру списку:
resize(): Модуль collections містить клас deque, який дозволяє ефективно змінювати розмір списку з обох кінців.

```
from collections import deque
my_list = deque([1, 2, 3])
my_list.appendleft(0)
# [0, 1, 2, 3]
my_list.append(4)
# [0, 1, 2, 3, 4]
```

Злиття списків:

`extend()`: З'єднує (розширює) один список іншим.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1.extend(list2)
# [1, 2, 3, 4, 5, 6]
```

Створення списку з повторюваних значень:

*: Створює список, повторюючи значення.

```
repeated_list = [0] * 5
# [0, 0, 0, 0, 0]
```

Захист від видалення елемента, якщо не існує:

`remove()`: Можна використовувати конструкцію if елемент in список для перевірки наявності елемента перед його видаленням.

```
my_list = [1, 2, 3, 4, 5]
element_to_remove = 6 if element_to_remove in my_list:
my_list.remove(element_to_remove)
```

Отримання копії списку без зміни оригіналу:

`list()` або зріз `[:]`: Щоб отримати копію списку без зміни оригіналу, можна використовувати функцію list() або зріз [:].

```
original_list = [1, 2, 3]
copied_list = list(original_list)
# Або copied_list = original_list[:]
```

Застосування функції до кожного елемента списку:

`map()`: Можна застосувати функцію до кожного елемента списку за допомогою map().

```
def square(x): return x ** 2
my_list = [1, 2, 3, 4, 5]
squared_list = list(map(square, my_list))
# [1, 4, 9, 16, 25]
```

Фільтрація елементів списку:
`filter()`: Можна фільтрувати список, застосовуючи функцію-фільтр до кожного елемента.

```
def is_even(x): return x % 2 == 0
my_list = [1, 2, 3, 4, 5]
even_numbers = list(filter(is_even, my_list))
# [2, 4]
```

Списки - це потужний і розгалужений тип даних в Python, який надає безліч можливостей для зберігання та обробки даних.

СЛОВНИКИ

Словники в Python - це структура даних, яка дозволяє зберігати пари ключ-значення, де кожен ключ унікальний. Вони також називаються асоціативними масивами або хеш-картами.

Створення словника: створення словника, перерахувнням пари ключ-значення у фігурних дужках та розділяючи їх двокрапкою.

```
my_dict = {"name": "John", "age": 30, "city": "New
York"}
```

Додавання та зміна значень: додавання нових пари ключ-значення до словника або зміна значення, присвоюючи нове значення існуючому ключу.

```
my_dict["email"] = john@example.com
# Додавання нової пари ключ-значення
my_dict["age"] = 31
# Зміна значення ключа "age"
```

Видалення пари ключ-значення: видалення пари ключ-значення за допомогою ключа та за допомогою оператора del.

```
del my_dict["city"]
# Видалення пари "city": "New York"
```

Доступ до значень за ключами: отримання значення, звертаючись до словника за ключем.

```
name = my_dict["name"]
# Отримання значення, пов'язаного з ключем "name"
```

Перевірка наявності ключа: перевірка чи існує певний ключ у словнику за допомогою оператора in.

```
has_email = "email" in my_dict
# Перевірка наявності ключа "email" у словнику
```

Отримання списку ключів та значень:
`keys()`: Повертає список всіх ключів у словнику.
`values()`: Повертає список всіх значень у словнику.
`items()`: Повертає список кортежів (ключ, значення) у словнику.

```
keys_list = my_dict.keys()
# Список ключів
values_list = my_dict.values()
# Список значень
items_list = my_dict.items()
# Список кортежів (ключ, значення)
```

Зміна розміру словника:
`clear()`: Очищає вміст словника, залишаючи його порожнім.
`copy()`: Створює поверхневу копію словника.

```
my_dict.clear()
# Порожній словник
copied_dict = my_dict.copy()
# Поверхнева копія словника
```

Злиття словників:

`update()`: З'єднує (оновлює) один словник іншим, додаючи нові ключ-значення та оновлюючи існуючі.

```
dict1 = {"name": "John", "age": 30}
dict2 = {"city": "New York", "email":
"john@example.com"}
dict1.update(dict2)
# З'єднує словники
```

Словники - це важливий тип даних в Python для зберігання та доступу до даних за допомогою унікальних ключів. Вони дозволяють організовувати дані у вигляді пар ключ-значення і динамічно змінювати вміст.

Отримання значення за ключем з безпечним доступом:

`get()`: Метод get(key, default) дозволяє отримати значення, пов'язане з ключем key. Якщо ключ не існує в словнику, то повертається значення за замовчуванням default.

```
my_dict = {"name": "John", "age": 30}
name = my_dict.get("name", "Unknown")
# Отримання значення ключа "name"
address = my_dict.get("address", "Unknown")
# Отримання значення ключа "address" (за замовчуванням
"Unknown")
```

Видалення та отримання значення за ключем з однією операцією:

`pop()`: Метод pop(key) видаляє пару ключ-значення за ключем key та повертає значення цього ключа.

```
my_dict = {"name": "John", "age": 30}
name = my_dict.pop("name")
# Видаляє та повертає значення ключа "name"
```

Захист від помилки при доступі за ключем:

`setdefault()`: Метод setdefault(key, default) додає новий ключ зі значенням default, якщо ключ відсутній у словнику, інакше повертає існуюче значення ключа.

```
my_dict = {"name": "John", "age": 30}
address = my_dict.setdefault("address", "Unknown")
# Додає ключ "address" зі значенням "Unknown"
```

```
age = my_dict.setdefault("age", 31)
# Ключ "age" вже існує, тому повертає існуюче значення
(30)
```

Ітерація через ключі та значення: Ви можете ітерувати через словник, отримуючи доступ до ключів і значень.

```
my_dict = {"name": "John", "age": 30} for key in
my_dict: print(key, my_dict[key])
```

Зміна розміру словника (копіювання):
`copy()`: Можна скопіювати словник для створення поверхневої копії.

```
original_dict = {"name": "John", "age": 30}
copied_dict = original_dict.copy()
# Поверхнева копія словника
```

Зміна розміру словника (глибоке копіювання): Для глибокого копіювання словника (включаючи всі вкладені об'єкти), можна використовувати бібліотеку copy.

```
import copy
original_dict = {"name": "John", "age": 30}
deep_copied_dict = copy.deepcopy(original_dict)
# Глибока копія словника
```

Сортування словника за ключами або значеннями:
`sorted()`: Можна відсортувати словник за ключами або значеннями.

```
my_dict = {"name": "John", "age": 30, "city": "New
York"}
sorted_dict_by_keys = dict(sorted(my_dict.items()))
# Сортування за ключами
sorted_dict_by_values = dict(sorted(my_dict.items(),
key=lambda item: item[1]))
# Сортування за значеннями
```

Заповнення словника значеннями за замовчуванням:
`fromkeys()`: Метод fromkeys() створює новий словник зі списком ключів і встановлює для всіх ключів одне значення за замовчуванням.

```
keys = ["name", "age", "city"]
default_value = "Unknown" my_dict = dict.fromkeys(keys, default_value)
# {"name": "Unknown", "age": "Unknown", "city": "Unknown"}
```

Словники - потужний тип даних в Python для зберігання та роботи з пар ключ-значення. Вони широко використовуються для представлення асоціативних даних і надають можливість ефективної роботи з ними.

Також до словників можна застосовувати наступні операції:

- Оператори рівності та нерівності.
- Оператори тотожності.
- Логічні оператори (кон’юнкція, диз’юнкція).

У всіх словників можна викликати стандартні вбудовані методи, інформацію про які можна знайти наступним чином: `print(help(dict))`

КОРТЕЖИ

Кортежі в Python - це інший тип даних для зберігання об'єктів, але, на відміну від списків, кортежі є незмінюваними, тобто їхні елементи не можуть бути змінені після створення.

Створення кортежу: Кортеж можна створити, перераховуючи елементи у круглих дужках та розділяючи їх комами.

```
my_tuple = (1, 2, 3, 4, 5)
```

Доступ до елементів кортежу за індексом: доступ до елементів кортежу, вказуючи їх індекси (індексація починається з 0).

```
first_element = my_tuple[0]
# Перший елемент
last_element = my_tuple[-1]
# Останній елемент
```

Зрізи кортежу (Slicing): За допомогою зрізів отримують підкортеж з вихідного кортежу.

```
sub_tuple = my_tuple[1:4]
# Елементи з індексами 1, 2, 3
```

Довжина кортежу: визначення кількість елементів у кортежі (функції len()).

```
length = len(my_tuple)
# Кількість елементів у кортежі
```

Конкатенація кортежів: обєднання двох або більше кортежів (оператор +).

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
concatenated_tuple = tuple1 + tuple2
# (1, 2, 3, 4, 5, 6)
```

Повторення кортежу: Створення нового кортежу, повторивши існуючий декілька разів.

```
repeated_tuple = my_tuple * 3
# (1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

Перевірка наявності елемента в кортежі: використання оператор in для перевірки, чи міститься певний елемент у кортежі.

```
is_present = 3 in my_tuple
# Перевірка, чи міститься число 3 у кортежі
```

Змішування кортежу: Модуль random може використовуватися для випадкового перемішування кортежу, змінюючи порядок його елементів.

```
import random
my_tuple = (1, 2, 3, 4, 5)
shuffled_tuple = random.sample(my_tuple, len(my_tuple))
# Випадкове перемішування кортежу
```

Копіювання кортежу: Кортежи є незмінюваними, тому їхні елементи не можуть бути змінені після створення. Для створення копії можна використовувати той самий кортеж.

```
original_tuple = (1, 2, 3)
copied_tuple = original_tuple
# Копія кортежу
```

Це основні операції, які можна виконувати над кортежами в Python. Кортежі корисні в тих випадках, коли вам потрібно створити незмінюваний набір даних, і вони зазвичай використовуються там, де важлива стійкість до змін.

МНОЖИНИ

Множини в Python - це колекція унікальних та незмінних об'єктів. Множини використовуються для виконання різних операцій над множинами, такі як об'єднання, перетин, різниця та інші.

Створення множини: Множину можна створити, перераховуючи її елементи у фігурних дужках та розділяючи їх комами.

```
my_set = {1, 2, 3, 4, 5}
```

Також можна створити множину за допомогою функції `set()`:

```
my_set = set([1, 2, 3, 4, 5])
```

Додавання та видалення елементів:
`add()`: Додає елемент до множини.

```
my_set.add(6)
# Додає число 6 до множини
```

`remove()`: Видаляє елемент із множини. Якщо елемент відсутній, видає помилку.

```
my_set.remove(3)
# Видаляє число 3 із множини
```

`discard()`: Видаляє елемент із множини, але не видає помилку, якщо елемент відсутній.

```
my_set.discard(7)
# Видаляє число 7, але не видає помилку, якщо він відсутній
```

Перевірка наявності елемента в множині: можна використовувати оператор `in` для перевірки наявності елемента в множині.

```
has_element = 4 in my_set
# Перевірка наявності числа 4 у множині
```

Об'єднання множин:
`union()`: Об'єднує дві множини та створює нову множину, яка містить всі унікальні елементи з обох множин.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1.union(set2)
# {1, 2, 3, 4, 5}
```

Можна також використовувати оператор | для об'єднання множин:

```
union_set = set1 | set2
# {1, 2, 3, 4, 5}
```

Перетин множин:
`intersection()`: Знаходить елементи, які містяться у обох множинах, і створює нову множину.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection_set = set1.intersection(set2)
# {3}
```

Можна також використовувати оператор & для знаходження перетину множин:

```
intersection_set = set1 & set2
# {3}
```

Різниця множин:
`difference()`: Знаходить елементи, які містяться у першій множині, але не містяться у другій, і створює нову множину.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
difference_set = set1.difference(set2)
# {1, 2}
```

Можна також використовувати оператор - для знаходження різниці множин:

```
difference_set = set1 - set2 # {1, 2}
```

Симетрична різниця множин:
`symmetric_difference()`: Знаходить елементи, які містяться тільки в одній з множин, і створює нову множину.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
symmetric_difference_set =
set1.symmetric_difference(set2)
# {1, 2, 4, 5}
```

Можна також використовувати оператор ^ для знаходження симетричної різниці множин:

```
symmetric_difference_set = set1 ^ set2
# {1, 2, 4, 5}
```

Множини в Python надають операції для роботи з унікальними значеннями та множинами даних. Вони корисні для видалення дублікатів, знаходження спільних елементів між колекціями і багатьох інших завдань.
Порожня множина: створення порожньої множини, (функція set()):

```
empty_set = set()
```

Копіювання множини: Множини є змінними об'єктами, тому, якщо ви просто надаєте ім'я іншій множині, ви створюєте посилання на ту саму множину. Для створення поверхневої копії множини використовують метод copy().

```
original_set = {1, 2, 3}
copied_set = original_set.copy()
# Поверхнева копія множини
```

Перевірка підмножини та надмножини:
`issubset()`: Перевіряє, чи є одна множина підмножиною іншої.
`issuperset()`: Перевіряє, чи є одна множина надмножиною іншої.

```
set1 = {1, 2, 3, 4, 5}
set2 = {2, 4}
is_subset = set2.issubset(set1)
# Перевірка, чи set2 є підмножиною
set1 is_superset = set1.issuperset(set2)
# Перевірка, чи set1 є надмножиною set2
```

Видалення елементів із множини за умовою:
`remove()`: Видаляє елемент із множини, але видає помилку, якщо елемент відсутній.

```
my_set = {1, 2, 3, 4, 5}
my_set.remove(6)
```

`discard()`: Видаляє елемент із множини, але не видає помилку, якщо елемент відсутній.

```
my_set.discard(6)
# Видаляє число 6 (без помилки, якщо відсутнє)
```

Очищення множини:
clear(): Очищає всі елементи множини, залишаючи її порожньою.

```
my_set.clear()
# Очищення множини
```

Конвертація множини в список та навпаки: можна конвертувати множину в список та навпаки за допомогою функцій list() і set().

```
my_set = {1, 2, 3}
my_list = list(my_set)
# Конвертація множини в список
new_set = set(my_list)
# Конвертація списку в множину
```

Ці операції дозволяють вам ефективно працювати з множинами в Python і виконувати різні операції над ними, такі як додавання, видалення та порівняння множин. Множини особливо корисні, коли потрібно зберігати унікальні значення і виконувати операції над ними.

Також до множин можна застосовувати наступні операції:

- Оператори рівності та нерівності.
- Оператори тотожності.
- Оператори належності.
- Логічні оператори.

У всіх множин та незмінних множини можна викликати стандартні вбудовані методи, інформацію про які можна знайти наступним чином: `print(help(set))`, `print(help(frozenset))`.

МАСИВИ

У Python, модуль array надає простий спосіб створення та керування масивами, які є послідовностями елементів одного типу даних. Масиви є корисними, коли потрібно ефективно зберігати і обробляти дані одного типу.

Імпорт модуля array:

```
from array import array
```

Створення масиву: Щоб створити масив, потрібно вказати тип даних елементів масиву. Для прикладу, тип "i" вказує на цілі числа. Можливі типи даних включають 'i' (цілі числа), 'f' (змінні з плаваючою комою) і так далі.

```
int_array = array('i', [1, 2, 3, 4, 5])
```

Додавання та видалення елементів:

append(): Додає новий елемент в кінець масиву.

```
int_array = array('i', [1, 2, 3, 4, 5])
int_array.append(6)
# Додає число 6 в кінець масиву
```

extend(): Розширює масив іншими елементами з іншого ітерабельного об'єкту (списку, кортежу, іншого масиву тощо).

```
other_array = array('i', [6, 7, 8])
int_array.extend(other_array)
# Додає елементи із other_array в кінець int_array
```

pop(): Видаляє та повертає елемент за певним індексом.

```
removed_element = int_array.pop(3)
# Видаляє та повертає четвертий елемент (індекс 3)
```

Доступ до елементів масиву: Ви можете отримувати доступ до елементів масиву за індексом.

```
element = int_array[2]
# Отримання третього елементу (індекс 2)
```

Зміна елементів масиву: Елементи масиву можна змінювати, просто присвоюючи їм нові значення.

```
int_array[2] = 10
# Змінює третій елемент на 10
```

Довжина масиву: можна визначити кількість елементів у масиві за допомогою функції len().

```
length = len(int_array)
# Кількість елементів у масиві
```

Пошук елементів у масиві:

index(): Знаходить індекс першого входження певного значення.

```
index = int_array.index(3)
# Знаходить індекс першого входження числа 3
```

count(): Підраховує кількість входжень певного значення у масив.

```
count = int_array.count(2)
# Підраховує кількість входжень числа 2
```

Видалення елементів із масиву за значенням:
`remove()`: Видаляє перше входження певного значення.

```
int_array.remove(4)
# Видаляє перше входження числа 4
```

Сортування масиву:

`sort()`: Сортує масив за зростанням.

```
int_array = array('i', [4, 1, 3, 2, 5])
int_array.sort()
# Сортує масив
```

Зворотне сортування масиву:
`reverse()`: Змінює порядок елементів на обернений.

```
int_array.reverse()
# Змінює порядок елементів на обернений
```

Масиви з модуля array надають ефективний спосіб зберігання та обробки даних одного типу. Вони особливо корисні, коли потрібно оптимізувати роботу з пам'яттю та працювати з великими масивами чисел.
Копіювання масиву:

```
copied_array = int_array.copy() # Глибока копія масиву
```

Очищення масиву:
`clear()`: Очищає всі елементи масиву, залишаючи його порожнім.

```
int_array.clear() # Очищення масиву
```

Злиття масивів: Ви можете об'єднати два масиви, використовуючи оператор +:

```
array1 = array('i', [1, 2, 3])
array2 = array('i', [4, 5, 6])
merged_array = array1 + array2
# Об'єднання двох масивів
```

Порівняння масивів: Можна порівнювати масиви за допомогою операторів порівняння, таких як ==, !=, <, <=, >, >=. Порівняння виконується елемент за елементом.

```
array1 = array('i', [1, 2, 3])
array2 = array('i', [1, 2, 3])
is_equal = array1 == array2
# Порівняння двох масивів за рівністю
```

Конвертація масиву в список та навпаки: можна конвертувати масив в список та навпаки за допомогою методів tolist() та fromlist():

```
int_array = array('i', [1, 2, 3, 4, 5])
list_representation = int_array.tolist()
# Конвертація масиву в список
new_array = array('i')
new_array.fromlist(list_representation)
# Конвертація списку в масив
```

Модуль array в Python надає зручний спосіб створення, зберігання та роботи з масивами даних одного типу. Вони корисні в задачах, де важливо забезпечити ефективну роботу з пам'яттю та працювати з числовими даними.

ДВОБІЧНІ ЧЕРГИ.

Двобічна черга (англ. double-ended queue, або deque) - це спеціальна структура даних, яка дозволяє додавати та видаляти елементи як з початку, так і з кінця черги. У Python для роботи з двобічними чергами використовується модуль collections.deque. Він надає ефективні операції додавання та видалення елементів як на початку, так і в кінці черги.

Імпорт модуля та створення двобічної черги:

```
from collections import deque
# Створення двобічної черги
my_deque = deque()
```

Додавання елементів:

`append()`: Додає елемент в кінець черги.

```
my_deque.append(1)
# Додає число 1 в кінець черги  appendleft(): Додає
елемент на початок черги.
my_deque.appendleft(0)
# Додає число 0 на початок черги
```

Видалення елементів:

`pop()`: Видаляє та повертає елемент із кінця черги.

```
popped_element = my_deque.pop()
# Видаляє та повертає останній елемент
```

`popleft()`: Видаляє та повертає елемент із початку черги.

```
popped_element = my_deque.popleft()
# Видаляє та повертає перший елемент
```

Отримання першого та останнього елементів:

[-1]: Отримання останнього елемента черги.

```
last_element = my_deque[-1]
# Останній елемент черги
```

[0]: Отримання першого елемента черги.

```
first_element = my_deque[0]
# Перший елемент черги
```

Розмір черги:

`len()`: Отримання кількості елементів у черзі.

```
size = len(my_deque)
# Кількість елементів у черзі
```

Перевірка наявності елементів у черзі: можна використовувати оператор in для перевірки наявності елемента в черзі.

```
has_element = 5 in my_deque
# Перевірка наявності числа 5 у черзі
```

Двобічні черги дуже корисні, коли потрібно виконувати ефективні операції додавання та видалення елементів як на початку, так і в кінці структури даних. Вони використовуються в різних ситуаціях, включаючи реалізацію черги, стеку та інших завдань, де важливо зберігати порядок елементів.

Видалення всіх елементів черги:

`clear()`: Очищає всі елементи черги, залишаючи її порожньою.

```
my_deque.clear()
# Очищення черги
```

Створення двобічної черги із ітерабельного об'єкту: можна створити двобічну чергу, передавши ітерабельний об'єкт, такий як список чи кортеж, у конструктор deque:

```
iterable = [1, 2, 3, 4, 5]
my_deque = deque(iterable)
# Створення черги із ітерабельного об'єкту
```

Зміна розміру черги: можна встановити максимальний розмір черги за допомогою методу maxlen. Це дозволяє обмежити кількість елементів у черзі.

```
my_deque = deque(maxlen=3)
# Черга із максимальним розміром 3
my_deque.extend([1, 2, 3, 4, 5])
# Додає більше елементів
```

У цьому випадку черга буде обмежена трьома останніми елементами, а найстарший елемент буде видалятися при додаванні нового елемента.

Обрізка черги: Метод rotate() дозволяє обрізати чергу, переміщуючи елементи ліворуч або праворуч.

```
my_deque = deque([1, 2, 3, 4, 5])
my_deque.rotate(2)
# Повертає два останні елементи на початок черги
```

Після цього вміст черги буде [4, 5, 1, 2, 3].

Ці операції дозволяють ефективно працювати з двобічними чергами в Python і виконувати різні операції з додаванням, видаленням та опрацюванням елементів як на початку, так і в кінці черги. Двобічні черги корисні для різноманітних завдань, включаючи обробку даних у порядку FIFO (перший прийшов, перший обслужений) та LIFO (останній прийшов, перший обслужений).

УПОРЯДКОВАНІ СПИСКИ.

Упорядкований список - це структура даних, в якій елементи розміщені в певному порядку. У Python, ви можете створити упорядковані списки, використовуючи стандартні списки та функції для сортування та обробки елементів. Ось деякі основні операції та методи для роботи з упорядкованими списками:

Додавання елементів в упорядкований список: щоб додати новий елемент в упорядкований список, можна використовувати методи append() або insert() для додавання елементу на відповідну позицію так, щоб список залишався упорядкованим.

```
ordered_list = [1, 3, 5, 7]
new_element = 4
# За допомогою append() - додаємо та сортуємо вручну
ordered_list.append(new_element)
ordered_list.sort()
# За допомогою insert() - вставляємо на відповідну
позицію
index_to_insert = bisect_left(ordered_list,
new_element)
ordered_list.insert(index_to_insert, new_element)
```

Видалення елементів із упорядкованого списку:

`remove()`: Видаляє перше входження певного значення.

```
ordered_list = [1, 2, 3, 4, 5]
```

```
ordered_list.remove(3)
# Видалення першого входження числа 3
```

`pop()`: Видаляє та повертає елемент за певним індексом.

```
removed_element = ordered_list.pop(2)
# Видалення та повертає третій елемент (індекс 2)
```

Доступ до елементів упорядкованого списку: можна отримувати доступ до елементів за індексом, так само, як в звичайному списку.

```
element = ordered_list[1]
# Отримання другого елемента (індекс 1)
```

Зміна значення елементів упорядкованого списку: Елементи упорядкованого списку можуть бути змінені, просто присвоївши їм нові значення.

```
ordered_list[0] = 10
# Змінює перший елемент на 10
```

Перевірка наявності елемента упорядкованому списку: можна використовувати оператор in для перевірки наявності елемента упорядкованому списку.

```
has_element = 5 in ordered_list
# Перевірка наявності числа 5
```

Пошук елементів упорядкованого списку:
`index()`: Знаходить індекс першого входження певного значення.

```
index = ordered_list.index(4)
# Знаходить індекс першого входження числа 4
```

`count()`: Підраховує кількість входжень певного значення в упорядкованому списку.

```
count = ordered_list.count(3)
# Підраховує кількість входжень числа 3
```

Сортування упорядкованого списку: упорядкований список вже впорядкований, тому не потрібно виконувати додаткові операції сортування.

Злиття двох упорядкованих списків: можна злити два упорядкованих списки в один, використовуючи операцію + або методи, такі як extend().

```
list1 = [1, 3, 5]
list2 = [2, 4, 6]
merged_list = list1 + list2
# Злиття двох списків
```

Упорядковані списки корисні для зберігання та обробки даних у відсортованому порядку. Вони використовуються в різних завданнях, включаючи пошук, фільтрацію та виконання операцій, які вимагають збереження порядку елементів.

Зрізання упорядкованого списку: можна створити зріз (підсписок) упорядкованого списку за допомогою зрізування.

```
ordered_list = [1, 2, 3, 4, 5]
sliced_list = ordered_list[1:4]
# Створює зріз [2, 3, 4]
```

Знаходження мінімального та максимального елементів: можна використовувати функції min() та max() для знаходження найменшого та найбільшого елементів упорядкованого списку.

```
ordered_list = [1, 2, 3, 4, 5]
min_value = min(ordered_list)
# Знаходження найменшого елемента
max_value = max(ordered_list)
# Знаходження найбільшого елемента
```

Зміна порядку елементів: можна використовувати функцію reverse() для зміни порядку елементів упорядкованого списку на обернений.

```
ordered_list = [1, 2, 3, 4, 5]
ordered_list.reverse()
# Змінює порядок елементів на обернений
```

Клонування упорядкованого списку: Для створення глибокої копії упорядкованого списку використовуйте метод copy() або зрізання.

```
ordered_list = [1, 2, 3, 4, 5]
cloned_list = ordered_list.copy()
# Глибока копія списку
```

Знайдення індексу для вставки нового елемента: Для вставки нового елемента упорядковано, можна використовувати функцію bisect_left() або bisect_right() з модуля bisect. Вони допомагають знайти позицію, на яку потрібно вставити елемент, щоб список залишився упорядкованим.

```
from bisect import bisect_left
ordered_list = [1, 3, 5, 7]
new_element = 4
index_to_insert = bisect_left(ordered_list,
new_element)
ordered_list.insert(index_to_insert, new_element)
```

Упорядковані списки корисні в багатьох випадках, де важливо зберігати та опрацьовувати дані в певному порядку. Вони забезпечують швидкий доступ до елементів та можливість виконання ефективних операцій пошуку та сортування.

ЧЕРГИ З ПРІОРИТИЕАМИ

Черги з пріоритетами - це структура даних, яка дозволяє зберігати та опрацьовувати елементи з пріоритетами, де елементи з вищим пріоритетом обробляються першими. У Python таку функціональність можна реалізувати, наприклад, за допомогою черги, яку забезпечує модуль queue, та використовувати структури даних для зберігання пар "елемент-пріоритет". Ось деякі операції та методи для роботи з чергами з пріоритетами в Python:

Імпорт модуля queue та створення черги з пріоритетами:

```
from queue import PriorityQueue
priority_queue = PriorityQueue()
```

Додавання елементів у чергу з пріоритетами: можна додавати елементи в чергу з вказанням їх пріоритету.

```
priority_queue.put((3, "Елемент 1"))
# Додавання елемента з пріоритетом 3
priority_queue.put((1, "Елемент 2"))
# Додавання елемента з пріоритетом 1
```

Видалення елементів із черги з пріоритетами:
`get()`: Видаляє та повертає елемент з найвищим пріоритетом.

```
item = priority_queue.get()
# Видаляє та повертає елемент з найвищим пріоритетом
```

Отримання розміру черги: можна отримати кількість елементів у черзі за допомогою методу qsize().

```
size = priority_queue.qsize()
# Кількість елементів у черзі
```

Перевірка наявності елементів у черзі: можна використовувати метод empty() для перевірки, чи є черга порожньою.

```
is_empty = priority_queue.empty()
# Перевірка, чи черга порожня
```

Зміна пріоритету елементів у черзі: Якщо потрібно змінити пріоритет певного елемента, потрібно видалити його та додати знову з новим пріоритетом.

```
item_to_change = (3, "Змінений елемент")
priority_queue.put(item_to_change)
# Додаємо знову з іншим пріоритетом
```

Зрізання черги з пріоритетами: Черги з пріоритетами не підтримують зрізання, оскільки це може порушити принцип роботи з пріоритетами.

Черги з пріоритетами корисні в різних сценаріях, де потрібно виконувати операції в порядку пріоритету. Вони забезпечують ефективну реалізацію завдань, де важливо зберігати та опрацьовувати елементи в порядку важливості.

Перегляд елементів черги з пріоритетами: Щоб переглянути всі елементи черги без їх видалення, можна використовувати цикл.

```
while not priority_queue.empty():
        item = priority_queue.get()
        print(item)
```

Це дозволить вам переглянути всі елементи у порядку їх пріоритету без їх видалення з черги.

Використання користувацьких об'єктів з пріоритетами: можна використовувати будь-які користувацькі об'єкти з пріоритетами, просто передавши кортеж у чергу, де перший елемент кортежу є пріоритетом. Це дозволяє зберігати та опрацьовувати складні об'єкти з пріоритетами.

Зберігання об'єктів у черзі з пріоритетами: можна зберігати об'єкти у черзі з пріоритетами, і вони будуть автоматично впорядковані за їхнім пріоритетом, якщо об'єкти правильно порівнюються між собою.

Ці операції та методи дозволяють ефективно керувати чергами з пріоритетами в Python і виконувати операції зберігання, видалення та отримання елементів в порядку їх важливості. Черги з пріоритетами корисні в багатьох сценаріях, включаючи планування завдань, обробку подій, прийом та обробку запитів з різними пріоритетами.

ТЕМА 3. ФУНКЦІЇ. ОБЛАСТІ ВИДИМОСТІ ТА ПРОСТОРИ ІМЕН. АРГУМЕНТИ ФУНКЦІЙ.

У Python функції є основною частиною програм, які виконують конкретні завдання та дозволяють вам структурувати та організувати код. Функції - це блоки коду, які можна викликати багато разів з різних частин програми. Ось загальна структура та основні поняття пов'язані з функціями в Python:

Оголошення функції: Функція оголошується за допомогою ключового слова def, після якого слідує ім'я функції та параметри в дужках.

```
def my_function(parameter1, parameter2):
     # Код функції
```

Параметри функції: Параметри - це значення, які передаються у функцію при її виклику. Вони визначаються у списку параметрів функції та використовуються всередині функції.

```
def add_numbers(x, y):
     result = x + y
     return result
```

Використання функції: Функцію можна викликати, передаючи їй значення параметрів у дужках. Функція може повертати значення за допомогою ключового слова return.

```
sum_result = add_numbers(3, 4)
print(sum_result) # Виведе 7
```

За замовчуванням параметри: можна визначити параметри функції зі значеннями за замовчуванням, що дозволяє викликати функцію без вказівки всіх параметрів.

```
def greet(name="Гість"):
	print(f"Привіт, {name}!")
greet()
# Виведе "Привіт, Гість!"
greet("Джон")
# Виведе "Привіт, Джон!"
```

Змінні області видимості: Змінні, оголошені всередині функції, мають локальну область видимості і не доступні поза функцією. Змінні, оголошені поза функцією, мають глобальну область видимості.

```
x = 10
def my_function():
	x = 5
	# Локальна змінна x
	print(x)
	# Виведе 5
my_function()
print(x)
# Виведе 10 (глобальна змінна x)
```

Змінні та параметри зі статичною областю видимості: Введені у Python 3.x, ключове слово nonlocal дозволяє змінювати значення змінних у вкладених функціях зі статичною областю видимості.

```
def outer_function():
	x = 10
	def inner_function():
		nonlocal x
		x = 5
	inner_function()
	print(x)
	# Виведе 5
outer_function()
```

Властивість return: Функція може повертати значення за допомогою ключового слова return. Якщо return відсутнє, функція повертає None.

```
def square(x):
	return x * x
result = square(4)
print(result)
# Виведе 16
```

Змінні довжини параметрів: Python дозволяє використовувати змінну кількість параметрів за допомогою *args та **kwargs.

```
def print_arguments(*args):
	for arg in args:
		print(arg)
print_arguments(1, 2, 3)
# Виведе 1, 2, 3
def print_keyword_arguments(**kwargs):
	for key, value in kwargs.items():
		print(f"{key}: {value}")
print_keyword_arguments(name="Джон", age=30)
# Виведе "name: Джон", "age: 30"
```

Лямбда-функції: Лямбда-функції - це анонімні функції, які можна визначити без імені.

```
square = lambda x: x * x
result = square(4)
print(result)
# Виведе 16
```

Документування функцій: Корисно документувати функції, щоб інші розробники зрозуміли, як вони працюють. Документацію функції зазвичай включають в рядок документації (docstring) всередині функції.

```
def greet(name):
	""" Функція приймає ім'я та виводить привітання.
	:param name: Ім'я для привітання. """
	print(f"Привіт, {name}!")
```

Функції в Python - це потужний механізм для організації та структурування коду. Вони дозволяють створювати

повторно використовувані блоки коду та зробити програму більш читабельною та обслуговуваною.

Рекурсія: Функції можуть викликати самі себе, що називається рекурсією. Рекурсивні функції корисні для вирішення завдань, які можуть бути розкладені на менші задачі.

Наприклад, ось рекурсивна функція для обчислення факторіалу:

```
def factorial(n):
	if n == 0:
		return 1
	 else:
		return n * factorial(n - 1)
```

Захист функцій: можна використовувати ключове слово pass для створення пустої функції. Це корисно, коли планується реалізувати функцію пізніше.

```
def placeholder_function():
	pass
```

Функції як аргументи інших функцій: У Python функції можуть бути передані як аргументи іншим функціям. Це корисно для створення загальних функцій, які працюють з різними функціями.

```
def apply_function(func, value):
	return func(value)
def double(x):
	return x * 2
result = apply_function(double, 5)
print(result)
# Виведе 10
```

Замикання (closures): Замикання виникають, коли функція визначається всередині іншої функції та має доступ до змінних об'єкта, що оточується.

```
def outer_function(x):
	def inner_function(y):
		return x + y
	return inner_function
closure = outer_function(10)
result = closure(5)
print(result)
# Виведе 15
```

Декоратори: Декоратори - це спосіб модифікувати функціональність іншої функції без зміни її коду. Вони дозволяють додавати додаткову логіку до функцій.

```
def my_decorator(func):
	def wrapper():
		print("Початок виконання")
		func()
		print("Завершення виконання")
	return wrapper
@my_decorator
def say_hello():
	print("Привіт!")
say_hello()
```

Функції грають ключову роль у структуруванні коду в Python та дозволяють створювати багато корисних конструкцій, таких як класи, модулі та бібліотеки. Вони допомагають зробити код більш організованим, читабельним та підтримуваним.

Генератори: Генератори - це спеціальний тип функцій, які використовують ключове слово yield для повернення значень по одному без зберігання їх у пам'яті. Генератори дозволяють ліниво обчислювати значення, що може бути корисно для обробки великих обсягів даних.

Ось приклад генератора, який генерує послідовність чисел Фібоначчі:

```
def fibonacci_generator():
	a, b = 0, 1
	while True:
	yield a
	a, b = b, a + b
```

Можна викликати генератор за допомогою циклу або функції next().

```
fib = fibonacci_generator()
for _ in range(10):
	print(next(fib))
```

Функції з анотаціями: Python дозволяє додавати анотації параметрів та повертаючих значень функцій. Це допомагає зрозуміти, як використовувати функцію та які типи даних вона приймає і повертає.

```
def add(a: int, b: int) -> int:
	return a + b
```

Анотації - це лише додаткова інформація для розробників та інструментів аналізу коду.

Аргументи за ключовим словом: можна викликати функції, передаючи аргументи за ключовим словом, що дозволяє змінювати порядок та об'єм параметрів.

```
def greet(name, message="Привіт"):
	print(f"{message}, {name}!")
greet("Джон")
# Виведе "Привіт, Джон!"
greet("Анна", message="Привіт, дорога")
# Виведе "Привіт, дорога, Анна!"
```

Інтерпретатор виразів eval() та exec(): можна використовувати функції eval() та exec() для виконання Python-коду, який подається у вигляді рядка. eval() використовується для обчислення виразів, а exec() для виконання блоків коду.

```
result = eval("3 + 4")
# Результат – 7
exec("print('Hello, World!')")
# Виведе "Hello, World!"
```

Функції як об'єкти першого класу: У Python функції є об'єктами першого класу, що означає, що можна

присвоювати їх змінним, передавати їх як аргументи та повертати їх як результати інших функцій.

```
def say_hello():
      print("Привіт!")
my_function = say_hello my_function()
# Виведе "Привіт!"
```

Функції в Python - це потужний механізм для організації та структурування коду. Вони дозволяють створювати короткі та повторно використовувані фрагменти коду, робити програми більш організованими та читабельними, а також вирішувати багато завдань у різних областях програмування.

ОБЛАСТІ ВИДИМОСТІ ТА ПРОСТОРИ ІМЕН

Області видимості та простори імен в мові програмування Python грають важливу роль в управлінні змінними, функціями та об'єктами в програмах. Давайте детальніше розглянемо ці поняття:

Простір імен (Namespace): Простір імен - це контейнер, де імена (змінні, функції, класи тощо) зберігаються та унікально ідентифікуються. Кожен об'єкт у Python має свій власний простір імен. Глобальні змінні та функції знаходяться в глобальному просторі імен, тоді як локальні змінні та функції існують в обмеженому просторі імен в межах функцій або методів.

Область видимості (Scope): Область видимості визначає, де конкретне ім'я (змінна, функція) може бути використане та де воно має значення. Є два основних типи областей видимості:

Глобальна область видимості (Global Scope): Змінні та функції, оголошені на глобальному рівні програми, доступні з будь-якого місця в програмі.

Локальна область видимості (Local Scope): Змінні та функції, оголошені всередині функцій або методів, є локальними та доступними лише в межах цих функцій.

Правило пошуку (LEGB): Python використовує правило LEGB для пошуку імен у різних областях видимості. Це означає наступне:

Local (Локальна): Пошук імен починається в локальній області видимості, де вони були визначені.

Enclosing (Навколишня): Якщо ім'я не знайдено в локальному просторі імен, пошук продовжується в навколишніх функціях, починаючи відзовну функцію та подальші вкладені функції.

Global (Глобальна): Якщо ім'я не знайдено в жодному з попередніх просторів імен, пошук відбувається в глобальному просторі імен.

Built-in (Вбудована): Якщо ім'я все ще не знайдено, пошук завершується в вбудованому просторі імен, де містяться вбудовані функції та об'єкти.

Оператори для роботи із змінними областями видимості: В Python є ключові слова, які дозволяють вам взаємодіяти з областями видимості. Наприклад, global вказує, що змінна має глобальну область видимості, а nonlocal дозволяє змінювати значення змінних в областях, навколишніх локальному.

Знання областей видимості та просторів імен важливо для правильного організації коду та уникнення конфліктів імен. Це допомагає підтримувати чистий та організований код та сприяє уникненню непередбачених помилок.

Аргументи функцій.

Аргументи функцій у мові програмування Python - це значення, які передаються в функцію під час її виклику. Функції в Python можуть приймати нуль або більше аргументів, і це дозволяє їм приймати вхідні дані для

подальшої обробки. Основні види аргументів у Python включають:

Позиційні аргументи (Positional Arguments): Це найпоширеніший тип аргументів. Позиційні аргументи передаються в тому порядку, в якому вони оголошені в функції.

```
def додавання(a, b):
	результат = a + b
	return результат
сума = додавання(3, 5)
# 3 - перший позиційний аргумент, 5 - другий позиційний аргумент
```

Аргументи за ключем (Keyword Arguments): У цьому варіанті ви можете передавати аргументи, вказуючи їхні імена. Це дозволяє передавати аргументи в будь-якому порядку та уникнути плутанини. \

```
def додавання(a, b):
	результат = a + b
	return результат
сума = додавання(b=5, a=3)
# Вказано аргументи за ключем
```

За замовчуванням (Default Arguments): можна задати значення за замовчуванням для аргументів у визначенні функції. Якщо значення для аргумента не передається при виклику функції, використовується значення за замовчуванням.

```
def вивід_повідомлення(повідомлення="Привіт, світ!"):
	print(повідомлення)
вивід_повідомлення()
# Виводиться "Привіт, світ!"
```

Зірочкові аргументи (*args): Зірочкові аргументи дозволяють функції приймати довільну кількість позиційних аргументів. Зірочка перед ім'ям аргумента дозволяє передавати багато значень у вигляді кортежу.

```
def сума_чисел(*args):
      результат = sum(args)
      return результат
сума = сума_чисел(1, 2, 3, 4, 5)
# Будь-яка кількість позиційних аргументів
```

****Аргументи зі зірочкою та ключові аргументи (*args та kwargs):** Ця комбінація дозволяє функціям приймати як позиційні, так і іменовані аргументи. Зірочка (*args) збирає позиційні аргументи, а дві зірочки (**kwargs) збирають ключові аргументи у словник.

```
def приклад(*args, **kwargs):
      print("Позиційні аргументи:", args)
      print("Ключові аргументи:", kwargs)
приклад(1, 2, 3, a=4, b=5)
```

Аргументи функцій дозволяють створювати загальні та зручні функції, які можуть працювати з різними вхідними даними. Можна використовувати різні комбінації аргументів для досягнення бажаної функціональності у своїх програмах.

ТЕМА 4: ІТЕРАТОРИ, ГЕНЕРАТОРИ ТА ФУНКЦІЇ-ГЕНЕРАТОРИ В PYTHON

У мові програмування Python ітератори та генератори є потужними інструментами для обробки послідовностей даних. Ця тема охоплює поняття ітераторів, генераторів та функцій-генераторів, а також вирази-генератори.

Ітератори (Iterators):

- Ітератори - це об'єкти, які дозволяють ітерувати (перебирати) послідовності даних, такі як списки, кортежі, словники і інші колекції.
- Ітератори реалізують два методи: `__iter__` і `__next__`. Метод `__iter__` повертає сам об'єкт

ітератора, а метод `__next__` повертає наступний елемент послідовності.

- Після вичерпання всіх елементів ітератор піднімає виняток StopIteration. Щоб легко ітерувати через ітератор, можна використовувати конструкцію `for ... in ...`.

Приклад:

```
my_list = [1, 2, 3, 4, 5]
my_iter = iter(my_list)
print(next(my_iter))  # 1
print(next(my_iter))  # 2
```

Генератори (Generators):

- Генератори - це спрощений спосіб створення ітераторів в Python. Вони визначаються за допомогою функції з ключовим словом `yield`.
- Функція-генератор виконується до першого виклику `yield`. Після цього вона призупиняє своє виконання і повертає значення, але зберігає свій стан.
- Після кожного наступного виклику `yield` функція-генератор продовжує виконання від точки паузи і виконується до наступного виклику `yield`.
- Генератори особливо корисні для обробки великих об'ємів даних, так як вони дозволяють генерувати дані "по запиту", не завантажуючи їх у пам'ять цілком.

Приклад функції-генератора:

```
python
def my_generator():
    yield 1
    yield 2
```

```
    yield 3

gen = my_generator()
for item in gen:
    print(item)
```

Вирази-генератори (Generator Expressions):

- Вирази-генератори - це однорядкові об'єкти, які генерують послідовності значень "ліниво", безпосередньо виразом.
- Вони схожі на спискові вирази, але генерують значення одне за одним, що зменшує витрати пам'яті.
- Вирази-генератори створюються за допомогою круглих дужок `()`.

Приклад виразу-генератора:

```
gen_expr = (x ** 2 for x in range(5))
for item in gen_expr:
    print(item)
```

У цій темі ми докладніше розглянемо кожен з цих концепцій та надамо більше прикладів використання ітераторів, генераторів та виразів-генераторів в Python.

Ітератори - це важливий аспект обробки послідовностей даних в Python. Вони надають можливість ефективно ітерувати (перебирати) елементи великих послідовностей без завантаження всіх елементів в пам'ять. Ось більш детальний огляд:

Ітератори і ітерабельні об'єкти

В Python ітератори реалізують два методи: `__iter__` і `__next__`. Об'єкти, які мають ці методи, називаються

ітераторами. Крім того, ітератори можуть бути використані з ітерабельними об'єктами.

- Метод `__iter__` повертає сам об'єкт ітератора (зазвичай `self`).
- Метод `__next__` повинен повертати наступний елемент послідовності, і він видає виняток `StopIteration`, коли всі елементи вже перебрані.

Приклад створення ітератора для ітерабельного об'єкта (списку):

```
class MyIterator:
    def __init__(self, max_value):
        self.max_value = max_value  # Максимальне
значення, до якого ітеруємо
        self.current = 0  # Поточне значення, з
якого починаємо ітерацію

    def __iter__(self):
        return self  # Повертаємо сам об'єкт як
ітератор

    def __next__(self):
        if self.current >= self.max_value:
            raise StopIteration  # Викидаємо
виняток StopIteration, коли досягли максимального
значення
        else:
            self.current += 1  # Збільшуємо
поточне значення на 1
            return self.current  # Повертаємо
поточне значення
```

Основні частини цього коду:

1. Конструктор `__init__` ініціалізує дві змінні: `max_value` (максимальне значення для ітерації) та `current` (поточне значення, з якого починається ітерація).

2. Метод `__iter__` повертає сам об'єкт `self` як ітератор. Це обов'язково для кожного ітератора.
3. Метод `__next__` визначає логіку ітерації. Він перевіряє, чи `current` менше або дорівнює `max_value`. Якщо так, він збільшує `current` на 1 та повертає нове значення `self.current`. Якщо `current` стає більшим або рівним `max_value`, викидається виняток `StopIteration`, щоб позначити завершення ітерації.

Можемо створити екземпляр `MyIterator` і використовувати його для ітерації:

```
my_iter = MyIterator(5)  # Створення екземпляра
ітератора із максимальним значенням 5
for item in my_iter:
    print(item)
```

Цей код створює об'єкт `my_iter`, і цикл `for` дозволяє ітерувати через `my_iter`. В результаті виводяться числа від 1 до 5, оскільки ітератор визначив ітерацію від 1 до `max_value`, яке в даному випадку дорівнює 5. Після досягнення 5 виникає виняток `StopIteration`, і ітерація завершується.

Переваги використання ітераторів:

- Ітератори дозволяють ефективно ітерувати через великі послідовності, такі як файли або потоки даних, без завантаження всієї послідовності в пам'ять.
- Вони особливо корисні при роботі з безкінечними послідовностями, такими як поток даних зі сенсорів або великі дані.

Вбудовані ітератори в Python**:** Python має вбудованих ітератори для роботи зі стандартними колекціями, такими як списки, кортежі, словники та інші.

Ось приклад використання ітератора для списку:

```
python
my_list = [1, 2, 3, 4, 5]
iter_list = iter(my_list)

print(next(iter_list))  # Виведе 1
print(next(iter_list))  # Виведе 2
```

За допомогою вбудованих ітераторів ви можете зручно перебирати елементи великих колекцій.

Ітератори дозволяють ефективно працювати з послідовностями даних в Python, полегшуючи ітерацію через великі або безкінечні послідовності та споживаючи менше пам'яті.

Зважаючи на широкий спектр використання ітераторів в Python, наведемо кілька прикладів:

Ітерація через файли: Однією з типових ситуацій є читання великих файлів, де завантаження усього вмісту в пам'ять недоцільно. Python дозволяє легко читати файли рядок за рядком за допомогою ітераторів:

```
with open('large_file.txt', 'r') as file:
  for line in file:
    process_line(line)
```

В цьому прикладі `file` є ітератором, який читає файл рядок за рядком.

Ітерація через словники: Можна ітерувати через ключі та значення словника:

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
for key in my_dict:
    print(key, my_dict[key])
```

Або можна використовувати методи словника, які повертають ітератори:

```
for key in my_dict.keys():
    print(key)

for value in my_dict.values():
    print(value)

for key, value in my_dict.items():
    print(key, value)
```

Ітерація через послідовності чисел: Python надає функцію `range`, яка дозволяє створювати послідовності чисел для ітерації:

```
python
for i in range(5):
    print(i)  # Виведе числа від 0 до 4
```

Це корисно, коли вам потрібно створити послідовність чисел для ітерації.

Власний ітератор із класу: Також можемо створити свій власний ітератор.

```
class MyIterator:
    def __init__(self, max_value):
        self.max_value = max_value
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current >= self.max_value:
```

```
            raise StopIteration
        else:
            self.current += 1
            return self.current
```

Ітератори корисні там де потрібно ефективно перебирати елементи послідовностей даних.

Генератори в Python

Генератори в Python - це потужний механізм, який дозволяє створювати ітератори за допомогою спеціальних функцій. Генератори особливо корисні для роботи з великими об'ємами даних, оскільки вони дозволяють генерувати значення "по запиту", не завантажуючи всю колекцію даних в пам'ять одразу. Давайте розглянемо детальніше, як створювати і використовувати генератори в Python.

1. Створення генераторів: Генератори визначаються за допомогою функцій, які містять ключове слово `yield`. Коли функція містить `yield`, вона стає функцією-генератором. Ось приклад простого генератора:

```
def simple_generator():
    yield 1
    yield 2
    yield 3
```

2. Використання генераторів: Для використання генераторів, ви створюєте екземпляр функції-генератора і викликаєте її для отримання ітератора. Використовуйте функцію `next()` для витягування значень з генератора. Ось приклад:

```
my_generator = simple_generator()
```

```
print(next(my_generator))  # Виводить 1
print(next(my_generator))  # Виводить 2
print(next(my_generator))  # Виводить 3
```

3. Автоматичний відповідь на вичерпання: Коли генератор вичерпує всі свої значення, він автоматично викликає виняток `StopIteration`. Ви можете також використовувати цикл `for` для ітерації через генератор:

```
my_generator = simple_generator()
for value in my_generator:
    print(value)  # Виводить 1, 2, 3
```

4. Генератори для оптимізації пам'яті: Генератори особливо корисні, коли вам потрібно обробляти великі об'єми даних. Вони дозволяють генерувати дані по одному значенню, звільняючи пам'ять після кожного значення. Наприклад, ви можете створити генератор для зчитування великого текстового файлу, обробки кожного рядка і звільнення пам'яті.

Зважаючи на широкий спектр можливостей генераторів, розглянемо декілька прикладів:

1. Генератор послідовності чисел Фібоначчі: Ось приклад генератора, який генерує послідовність чисел Фібоначчі.

```
def fibonacci_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib = fibonacci_generator()
for _ in range(10):
    print(next(fib))  # Виводить перші 10 чисел
Фібоначчі
```

2. Генератор для роботи з великими файлами: Генератори дозволяють зчитувати та обробляти великі файли по одному рядку, не завантажуючи весь файл в пам'ять.

```
def process_large_file(filename):
    with open(filename, 'r') as file:
        for line in file:
            yield line

for line in process_large_file('large_file.txt'):
    # Обробка кожного рядка файлу
    pass
```

3. Генератори з умовою: Ви можете визначити генератор, який генерує значення лише відповідно до певної умови.

```
def even_numbers(n):
    for i in range(n):
        if i % 2 == 0:
            yield i

evens = even_numbers(10)
for value in evens:
    print(value)  #Виводить парні числа від 0 до 8
```

4. Генератори виразів: Вирази-генератори дозволяють генерувати значення без визначення окремої функції-генератора.

```
squares = (x ** 2 for x in range(5))
for square in squares:
  print(square)#Виводить квадрати чисел від 0 до 4
```

Генератори - потужний інструмент в Python для оптимізації роботи з послідовностями даних і оптимізації використання пам'яті, що стає надзвичайно корисним для роботи з великими об'ємами даних.

Вирази-генератори (Generator Expressions)

Вирази-генератори (Generator Expressions) - це зручний спосіб створення генераторів на основі виразів, які генерують послідовності значень "ліниво", безпосередньо виразом. Вони дуже схожі на спискові вирази, але генерують значення одне за одним, що дозволяє ефективно керувати пам'яттю. Розглянемо детальніше із прикладами.

1. Простий приклад генератора з числами:

```
gen = (x for x in range(5))
for num in gen:
    print(num)  # Виводить числа від 0 до 4
```

2. Фільтрація значень за певною умовою:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_gen = (x for x in numbers if x % 2 == 0)
for even_num in even_gen:
    print(even_num)  # Виводить парні числа від 2
до 10
```

3. Вирази-генератори з операціями:

```
data = [1, 2, 3, 4, 5]
squared_gen = (x ** 2 for x in data)
for squared in squared_gen:
    print(squared) #Виводить квадрати чисел від 1
до 5
```

4. Робота зі словниками:

```
prices = {'apple': 1.0, 'banana': 0.5, 'cherry':
2.0}
discounted_prices = {item: price * 0.9 for item,
price in prices.items()}
```

```
print(discounted_prices)  # Виводить словник із
знижеными цінами
```

5. Використання генератора для обчислення суми квадратів чисел:

```
numbers = [1, 2, 3, 4, 5]
sum_of_squares = sum(x ** 2 for x in numbers)
print(sum_of_squares)  # Виводить суму квадратів
чисел
```

6. Обчислення середнього значення списку чисел:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
average = sum(numbers) / len(numbers)
print(average)  # Обчислює середнє значення списку

# Використовуючи вираз-генератор
average = sum(x for x in numbers) / len(numbers)
print(average)  # Той самий результат, але з
виразом-генератором
```

7. Вибірка лише унікальних значень із списку:

```
data = [1, 2, 2, 3, 4, 4, 5, 5, 5]
unique_values = list(set(data))
print(unique_values)  # Виводить лише унікальні
значення

# Використовуючи вираз-генератор
unique_values = list({x for x in data})
print(unique_values)  # Той самий результат, але з
виразом-генератором
```

8. Об'єднання значень з двох списків:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined = list1 + list2
print(combined)  # Об'єднує значення з двох
списків
```

```
# Використовуючи вираз-генератор
combined = list(x for lst in (list1, list2) for x
in lst)
print(combined)  # Той самий результат, але з
виразом-генератором
```

9. Зведення до одного рядка списку слів:

```
words = ["Це", "приклад", "виразів-генераторів",
"в", "Python"]
sentence = ' '.join(words)
print(sentence)  # З'єднує слова в один рядок

# Використовуючи вираз-генератор
sentence = ' '.join(word for word in words)
print(sentence)  # Той самий результат, але з
виразом-генератором
```

Вирази-генератори дозволяють ефективно працювати з послідовностями даних і оптимізувати використання пам'яті, оскільки вони генерують значення "ліниво", по одному за раз.

Тема 5. Об'єкти. Класи. Класові ієрархії. Абстрактні класи.

Python підтримує багато різних типів даних, таких як числа, рядки та інші. Кожен з цих типів - це об'єкт, і кожен об'єкт має:

- тип
- внутрішнє представлення даних (примітивне або складне)
- набір процедур для взаємодії з об'єктом

Кожен об'єкт є екземпляром певного типу. Наприклад, число 1234 є екземпляром типу int, а рядок "привіт" є екземпляром типу рядок.

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ (ООП)

- ВСЕ В PYTHON Є ОБ'ЄКТОМ (і має тип)
- Можна створювати нові об'єкти певного типу
- Можна взаємодіяти з об'єктами
- Можна знищувати об'єкти
    - явно за допомогою del або просто «забути» про них
    - система Python звільнює знищені або недосяжні об'єкти - це називається "збором сміття".

ЩО ТАКЕ ОБ'ЄКТИ?

Об'єкти є абстракцією даних, яка включає:

(1) внутрішнє представлення через атрибути даних (2) інтерфейс для взаємодії з об'єктом через методи

(процедури/функції) - визначає поведінку, але приховує реалізацію

Яким чином списки представлені внутрішньо? Як список змінюється?

- Представлення списку внутрішньо: зв'язаний список елементів.
- Як змінити список? Використовуйте індексацію, зрізи, конкатенацію тощо.
- Використовуйте методи списків, такі як len(), min(), max(), del тощо.

ВНУТРІШНЄ ПРЕДСТАВЛЕННЯ МАЄ БУТИ ПРИВАТНИМ

Якщо ви працюєте з класами і об'єктами, зворотний доступ до внутрішнього представлення може порушити коректну поведінку. Внутрішнє представлення повинно бути прихованим.

ПЕРЕВАГИ ООП

- Упаковка даних в пакети разом з процедурами, що працюють з ними через чітко визначені інтерфейси.
- Розробка методом розділення та панування:
    - Реалізація та тестування поведінки кожного класу окремо.
    - Збільшена модульність, що зменшує складність.
- Класи сприяють повторному використанню коду:
    - Багато модулів Python визначають нові класи.
    - Кожен клас має своє власне середовище (не виникає конфлікту імен функцій).

- Наслідування дозволяє підкласам перевизначати або розширювати певний підмножину поведінки батьківського класу.

СТВОРЕННЯ І ВИКОРИСТАННЯ ВЛАСНИХ ТИПІВ ЗА ДОПОМОГОЮ КЛАСІВ

ВИЗНАЧЕННЯ ВЛАСНИХ ТИПІВ

Для створення нового типу використовується ключове слово "class". Наприклад:

```
class Coordinate:
    # Визначення атрибутів тут
```

Подібно до визначення функцій, код слід відступати для позначення того, що він належить до визначення класу.

Слово "object" вказує, що "Coordinate" є об'єктом Python і успадковує всі його атрибути. Тобто "Coordinate" є підкласом "object".

ЩО ТАКЕ АТРИБУТИ?

Атрибути - це дані та процедури, які "належать" класу:

- Дані атрибути: дані розглядаються як інші об'єкти, які складають клас. Наприклад, координата складається з двох чисел.
- Методи (процедурні атрибути): методи розглядаються як функції, які працюють лише з об'єктами класу. Вони визначають, як взаємодіяти з об'єктом. Наприклад, ви можете визначити метод "відстань" для обчислення відстані між двома об'єктами "Coordinate".

ВИЗНАЧЕННЯ СПОСОБУ СТВОРЕННЯ ЕКЗЕМПЛЯРА КЛАСУ

Спочатку потрібно визначити, як створювати екземпляр класу, використовуючи метод "**init**" для ініціалізації деяких атрибутів даних. Наприклад:

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

ФАКТИЧНЕ СТВОРЕННЯ ЕКЗЕМПЛЯРА КЛАСУ

Після визначення того, як створювати екземпляр класу, можна створювати нові екземпляри об'єктів, використовуючи спеціальний конструктор. Наприклад:

```
c = Coordinate(3, 4)
origin = Coordinate(0, 0)
print(c.x)
print(origin.x)
```

ЩО ТАКЕ МЕТОД?

Метод - це процедурний атрибут, схожий на функцію, яка працює лише з об'єктами цього класу. В Python завжди передається об'єкт як перший аргумент, та зазвичай його назву називають "self". Методи визначаються так само, як і функції, і можуть приймати параметри, виконувати операції та повертати значення.

ОГОЛОШЕННЯ МЕТОДУ ДЛЯ КЛАСУ Coordinate

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
    def distance(self, other):
        x_diff_sq = (self.x - other.x) ** 2
        y_diff_sq = (self.y - other.y) ** 2
        return (x_diff_sq + y_diff_sq) ** 0.5
```

ВИКОРИСТАННЯ МЕТОДУ

Ви можете використовувати методи, викликаючи їх на об'єкті класу. Наприклад:

```
c = Coordinate(3, 4)
zero = Coordinate(0, 0)
print(c.distance(zero))
```

ВИЗНАЧЕННЯ МЕТОДУ str

Ви можете визначити спеціальний метод "**str**" для представлення об'єкта у зручному для вас вигляді при використанні функції "print". Наприклад:

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        x_diff_sq = (self.x - other.x) ** 2
        y_diff_sq = (self.y - other.y) ** 2
    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

Для перевірити типу об'єкта використовують `type()`. Наприклад:

```
python
c = Coordinate(3, 4)
print(type(c))
```

також використовувати `isinstance()`, щоб перевірити, чи об'єкт є екземпляром певного класу:

- `print(isinstance(c, Coordinate))`

Спеціальні оператори включають `+`, `-`, `==`, `<`, `>`, `len()`, `print` та багато інших. Їх можна перевизначити для роботи з об'єктами класу за допомогою методів, які починаються та закінчуються подвійними підкресленнями. Наприклад:

- `__add__(self, other)` для оператора `self + other`
- `__sub__(self, other)` для оператора `self - other`
- `__eq__(self, other)` для оператора `self == other`
- `__lt__(self, other)` для оператора `self < other`
- `__len__(self)` для `len(self)`
- `__str__(self)` для функції `print(self)`

Ви можете створити новий тип для представлення чисел у вигляді дробів та визначити методи, які надають можливість додавати, віднімати, конвертувати у десятковий дріб, інвертувати дріб тощо.

У спеціальних методах можна перевизначити поведінку операторів та функцій для вашого класу. Наприклад, ви можете перевизначити метод `__add__`, щоб визначити операцію додавання (+) для об'єктів вашого класу.

Щоб визначити власний текстовий представник для об'єкта, використовуйте метод `__str__`. Наприклад, ви можете перевизначити цей метод для вашого класу

Coordinate, щоб при виведенні об'єкта за допомогою print отримати рядок з координатами.

Наприклад:

Розглянемо кожен із перевизначених методів більш детально.

1. `__add__(self, other)` для оператора `self + other`: Цей метод дозволяє визначити, як об'єкти класу Coordinate повинні поводитися при використанні оператора + для їх суми. Ось приклад:

```
python
def __add__(self, other):
    new_x = self.x + other.x
    new_y = self.y + other.y
    return Coordinate(new_x, new_y)
```

Зараз, коли ви додаєте два об'єкти Coordinate, вони будуть сумуватися як новий об'єкт Coordinate.

2. `__sub__(self, other)` для оператора `self - other`: Цей метод визначає поведінку віднімання одного об'єкта Coordinate від іншого. Ось приклад:

```
python
def __sub__(self, other):
    new_x = self.x - other.x
    new_y = self.y - other.y
    return Coordinate(new_x, new_y)
```

Це дозволить вам віднімати один об'єкт Coordinate від іншого.

3. `__eq__(self, other)` для оператора `self == other`: Цей метод дозволяє визначити, коли два

об'єкти `Coordinate` рівні один одному. Ось приклад:

```
python
def __eq__(self, other):
    return self.x == other.x and self.y == other.y
```

Цей метод перевіряє, чи обидва об'єкти `Coordinate` мають однакові значення `x` та `y`.

4. `__lt__(self, other)` для оператора `self < other`: Цей метод визначає порівняння двох об'єктів `Coordinate` для оператора < (менше). Ось приклад:

```
python
def __lt__(self, other):
    return (self.x ** 2 + self.y ** 2) < (other.x
** 2 + other.y ** 2)
```

Це порівнює об'єкти `Coordinate` за їх відстанню від початку координат.

5. `__len__(self)` для `len(self)`: Цей метод дозволяє визначити, як буде обчислюватися довжина об'єкта. Однак для об'єкта `Coordinate` це не дуже інтуїтивно. Можливо, було б корисніше мати метод для обчислення довжини списку координат.
6. `__str__(self)` для `print(self)`: Цей метод визначає, як об'єкт `Coordinate` повинен виводитися за допомогою функції `print()`. Ось приклад:

```
python
def __str__(self):
    return f"Coordinate: ({self.x}, {self.y})"
```

Цей метод дозволяє вам виводити об'єкти `Coordinate` у зрозумілому форматі при використанні функції `print()`.

За допомогою ООП можна упаковувати об'єкти, які мають спільні атрибути та процедури, і застосовувати абстракцію для розділення реалізації та використання об'єкта. Можна будувати структури об'єктів, які успадковують поведінку від інших класів об'єктів, і створювати власні класи на основі базових класів в Python.

Наслідування

Наслідування є однією з ключових концепцій об'єктно-орієнтованого програмування (ООП). Наслідування - це механізм, за допомогою якого новий клас (підклас або підклас) може успадковувати властивості і методи від іншого класу (базового класу або суперкласу). Це дозволяє створювати ієрархію класів, де підкласи можуть розширювати або змінювати функціонал базового класу.

В Python наслідування дозволяє створювати нові класи на основі існуючих класів, використовуючи їх властивості і методи. Клас, від якого інший клас успадковує, називається батьківським класом або суперкласом, а клас, який успадковує властивості і методи, називається дочірнім класом або підкласом.

Основні поняття наслідування в Python:

1. Батьківський клас (суперклас): Це клас, від якого інший клас успадковує властивості і методи.
2. Дочірній клас (підклас): Це клас, який успадковує властивості і методи від батьківського класу.
3. Поля і методи батьківського класу: Поля (атрибути) і методи, які інший клас успадковує від батьківського класу.

4. Перевизначення (поліморфізм): Дочірній клас може перевизначити (перевантажити) методи батьківського класу, якщо це необхідно.

Розглянемо приклад наслідування на прикладі класу "Coordinate" (координати), який ми створили раніше:

```
python
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        x_diff_sq = (self.x - other.x) ** 2
        y_diff_sq = (self.y - other.y) ** 2
        return (x_diff_sq + y_diff_sq) ** 0.5

    def __str__(self):
        return f"<{self.x},{self.y}>"
```

Тепер давайте створимо новий клас "ThreeDPoint" (точка в тривимірному просторі), який успадковуватиме властивості і методи від класу "Coordinate" та додамо до нього третю координату "z":

```
class ThreeDPoint(Coordinate):
    def __init__(self, x, y, z):
        super().__init__(x, y)  # Викликаємо
конструктор базового класу Coordinate
        self.z = z

    def distance(self, other):
        # Перевизначаємо метод distance для
тривимірної точки
        xy_distance = super().distance(other)  #
Викликаємо метод базового класу для обчислення
відстані в площині XY
        z_diff_sq = (self.z - other.z) ** 2
        return (xy_distance ** 2 + z_diff_sq) **
0.5
```

```
    def __str__(self):
        return f"<{self.x},{self.y},{self.z}>"
```

Основні аспекти наслідування в цьому прикладі:

1. Клас "ThreeDPoint" успадковує клас "Coordinate", вказавши його як суперклас у визначенні "class ThreeDPoint(Coordinate)".
2. Конструктор "ThreeDPoint" викликає конструктор базового класу "Coordinate" за допомогою "super().**init**(x, y)" для налаштування координат "x" та "y".
3. Клас "ThreeDPoint" має свою власну координату "z", яку не має базовий клас "Coordinate".
4. Метод "distance" в класі "ThreeDPoint" перевизначено так, щоб обчислити відстань у тривимірному просторі, додаючи координату "z". При цьому використовується метод базового класу "Coordinate" для обчислення відстані в площині XY.
5. Визначено метод "**str**", який генерує рядок для представлення об'єкта "ThreeDPoint" у зручному для виводу форматі.

Цей приклад демонструє, як за допомогою наслідування можна створити новий клас, який розширює функціональність базового класу та додає власну функціональність.

Побудуємо ієрархію об'єктів з трьох класів та додамо атрибут "колір" до них. Нехай у нас буде базовий клас " Coordinate " для точок двовимірному просторі, клас " ThreeDPoint " на основі базового класу "Point" для точок з координатами (x, y, z) із можливістю визначення відстані між ними, і клас "ColoredCoordinate" та

“ColorThreeDPoint” для точок з додатковим атрибутом "колір". Наприклад:

```
class Coordinate (object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        x_diff_sq = (self.x - other.x) ** 2
        y_diff_sq = (self.y - other.y) ** 2
        return (x_diff_sq + y_diff_sq) ** 0.5

class ColoredCoordinate(Coordinate):
    def __init__(self, x, y, color):
        super().__init__(x, y)
        self.color = color

    def __str__(self):
        return f"<{self.x},{self.y}>
({self.color})"

class ThreeDPoint(Point):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.z = z


    def distance(self, other):
        # Перевизначаємо метод distance для
тривимірної точки
        xy_distance = super().distance(other)  #
Викликаємо метод базового класу для обчислення
відстані в площині XY
        z_diff_sq = (self.z - other.z) ** 2
        return (xy_distance ** 2 + z_diff_sq) **
0.5

class ColorThreeDPoint(ThreeDPoint):
    def __init__(self, x, y, z, color):
        super().__init__(x, y, z)
        self.color = color
```

```
    def __str__(self):
        return f"<{self.x},{self.y},{self.z}>
({self.color})"


coordinate1 = Coordinate(2, 3)
coordinate2 = Coordinate(5, 6)

colored_coordinate1 = ColoredCoordinate(3, 4,
"червоний")
colored_coordinate2 = ColoredCoordinate(6, 7,
"синій")

three_d_point1 = ThreeDPoint(2, 2, 2)
three_d_point2 = ThreeDPoint(5, 5, 5)

color_three_d_point1 = ColorThreeDPoint(2, 2, 2,
"зелений")
color_three_d_point2 = ColorThreeDPoint(5, 5, 5,
"помаранчевий")

print(coordinate1.distance(coordinate2))
print(colored_coordinate1)
print(three_d_point1.distance(three_d_point2))
print(color_three_d_point1)
```

Розглянемо кожен клас детальніше.

1. Coordinate (Координата): Це базовий клас, який призначений для представлення точок в тривимірному просторі. Він містить два атрибути x та y, які представляють координати точки. Клас також має метод distance, який обчислює відстань між двома точками, використовуючи теорему Піфагора.
2. ColoredCoordinate (Координата з кольором): Цей клас успадковує від класу Coordinate і розширює його можливості. Він додає атрибут color, який визначає колір точки. Крім того, перевизначено метод __str__, щоб можна було зручно

представляти об'єкти цього класу у вигляді рядка. В рядку виводиться колір точки.
3. ThreeDPoint (Тривимірна точка): Цей клас також успадковує від класу Coordinate, але додає третю координату z. Крім того, він перевизначає метод distance для обчислення відстані між точками у тривимірному просторі.
4. ColorThreeDPoint (Тривимірна точка з кольором): Цей клас успадковує від класу ThreeDPoint та додає атрибут color, що представляє колір точки. Також перевизначено метод __str__ для зручного виводу точки у вигляді рядка разом з її кольором.

Абстрактний клас

Абстрактний клас у програмуванні є класом, який не може бути створений безпосередньо, тобто не може мати екземплярів. Його основна мета - визначити загальні характеристики та функціонал для підкласів, які будуть успадковувати цей клас. Абстрактні класи використовуються для надання загальної структури та методів, які мають бути реалізовані в підкласах. Вони виступають як шаблони, які допомагають забезпечити консистентність та стандартизацію в класах-нащадках.

Основні причини використання абстрактних класів:

1. Вимагання реалізації: Абстрактний клас може визначити методи, які мають бути реалізовані в підкласах. Це дозволяє вимагати від розробників підкласів реалізувати певні функції, щоб забезпечити правильну роботу програми.
2. Розділення загального функціоналу: Абстрактні класи дозволяють об'єднати спільний функціонал для різних класів, що реалізують певну концепцію.

Наприклад, ви можете мати абстрактний клас "Point", який містить загальний функціонал для всіх типів точок.

3. Поліморфізм: Абстрактні класи дозволяють використовувати поліморфізм, тобто можливість обробки об'єктів різних класів однаковим чином. Це дуже корисно при роботі з колекціями об'єктів різних класів.
4. Спрощення утримання: Абстрактні класи допомагають спростити процес утримання коду, оскільки вони дозволяють зосередитися на загальних аспектах функціоналу і уникнути дублювання коду в різних класах-нащадках.

Загальні методи абстрактного класу визначаються за допомогою декоратора `@abstractmethod`. Підкласи повинні реалізувати ці методи, інакше видасться помилка.

Абстрактні класи створюються з метою покращення структури програми, зробити її більш зрозумілою і підвищити її надійність.

```
from abc import ABC, abstractmethod

class Point(ABC):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @abstractmethod
    def distance(self, other):
        pass

class Coordinate(Point):
    def __init__(self, x, y):
        super().__init__(x, y)

    def distance(self, other):
        x_diff_sq = (self.x - other.x) ** 2
```

```
        y_diff_sq = (self.y - other.y) ** 2
        return (x_diff_sq + y_diff_sq) ** 0.5

class ColorCoordinate(Point):
    def __init__(self, x, y, color):
        super().__init__(x, y)
        self.color = color

    def __str__(self):
        return f"<{self.x},{self.y}>
({self.color})"

class ThreeDPoint(Point):
    def __init__(self, x, y, z, color):
        super().__init__(x, y)
        self.z = z
        self.color = color

    def distance(self, other):
        x_diff_sq = (self.x - other.x) ** 2
        y_diff_sq = (self.y - other.y) ** 2
        z_diff_sq = (self.z - other.z) ** 2
        return (x_diff_sq + y_diff_sq + z_diff_sq)
** 0.5

    def __str__(self):
        return f"<{self.x},{self.y},{self.z}>
({self.color})"

coordinate1 = Coordinate(2, 3)
coordinate2 = Coordinate(5, 6)
color_coordinate1 = ColorCoordinate(3, 4,
"червоний")
color_coordinate2 = ColorCoordinate(6, 7, "синій")
three_d_point1 = ThreeDPoint(2, 2, 2, "зелений")
three_d_point2 = ThreeDPoint(5, 5, 5,
"помаранчевий")

print(coordinate1.distance(coordinate2))
print(color_coordinate1)
print(three_d_point1.distance(three_d_point2))
```

Розглянемо кожен клас детальніше.

1. `Point` - абстрактний базовий клас, від якого успадковуються інші класи. Він має абстрактний метод `distance`, який буде визначено в підкласах. Цей клас містить атрибути `x` та `y`, які представляють координати точки.
2. `Coordinate` - клас, який успадковується від `Point`. Він реалізує метод `distance` для обчислення відстані між двома точками у двомірному просторі. Використовується формула Евклідової відстані для цього.
3. `ColorCoordinate` - ще один підклас `Point`, який додає атрибут `color` для представлення кольору точки. Він також має власний метод `__str__`, який повертає рядок з координатами і кольором точки.
4. `ThreeDPoint` - цей клас успадковується від `Point` і розширює його, додавши третю координату `z` і атрибут `color`. Метод `distance` реалізований для обчислення відстані між двома точками у тривимірному просторі, використовуючи формулу для тривимірної відстані.

Ці класи демонструють використання абстрактних класів для створення ієрархії класів і забезпечення спільного інтерфейсу для підкласів. Це полегшує розширення функціоналу і створення нових класів, які відповідають визначеному інтерфейсу.

Тема 6. ОБРОБКА ВИНЯТКІВ. МОДУЛІ ТА ПАКЕТИ. ДОКУМЕНТУВАННЯ КОДУ

Обробка винятків – це процес написання коду для перехоплення та обробки помилок чи винятків, які можуть виникати під час виконання програми. Це дозволяє розробникам створювати надійні програми, які продовжують працювати навіть у разі несподіваних подій чи помилок. Без системи обробки винятків це зазвичай призводить до фатальних збоїв.

Коли виникають винятки – Python шукає відповідний обробник винятків. Після цього, якщо оброблювач буде знайдено, виконується його код, в якому робляться доречні дії. Це може бути логування даних, виведення повідомлення, спроба відновити роботу програми після помилки. В цілому можна сказати, що обробка виключення допомагає підвищити надійність Python-застосунків, покращує можливості їх підтримки, полегшує їх налагодження.

Відмінності між оператором if та обробкою винятків

Головні відмінності між оператором **if** та обробкою винятків у Python виростають з їх цілей та сценаріїв використання.

Оператор **if** – це базовий будівельний елемент структурного програмування. Цей оператор перевіряє умову та виконує різні блоки коду, ґрунтуючись на тому, чи істинно перевіряється умова чи хибно. Наприклад:

```
     temperature = int(input("Please enter
temperature in Fahrenheit: "))
     if temperature > 100:
         print("Hot weather alert! Temperature
exceeded 100°F.")
     elif temperature >= 70:
         print("Warm day ahead, enjoy sunny
skies.")
```

```
    else:
        print("Bundle up for chilly temperatures.")
```

Обробка винятків, з іншого боку, відіграє важливу роль у написанні надійних та стійких до відмови програм. Ця роль розкривається через роботу з несподіваними подіями та помилками, які можуть виникати під час виконання програми.

Винятки використовуються для подачі сигналів про проблеми і для виявлення ділянок коду, які потребують поліпшення, налагодження або оснащення їх додатковими механізмами для перевірки помилок. Винятки дозволяють Python гідно справлятися із ситуаціями, у яких виникають помилки. У таких ситуаціях винятки дають можливість продовжувати виконання скрипту замість різко його зупиняти.

Розглянемо наступний код, що демонструє приклад того, як можна реалізувати обробку винятків та покращити ситуацію з потенційними відмовами, пов'язаними з розподілом на нуль:

```
def divide(x, y):
    result = x / y
    return result

result = divide(5, 0)
print(f"Result of dividing {x} by {y}: {result}")
```

```
Результат:
```

```
Traceback (most recent call last):
  File "<stdin>", line 8, in <module>
ZeroDivisionError: division by zero attempted
```

Після того, як було згенеровано виняток, програма, не дійшовши до інструкції **print**, відразу припиняє виконуватись.

Описаний виняток можна обробити, обернувши виклик функції divide в блок **try-except**:

```
def divide(x, y):
    result = x / y
    return result
try:
    result = divide(5, 0)
    print(f"Result of dividing {x} by {y}: {result}")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

```
Результат:

Cannot divide by zero.
```

Зробивши це, ми обробили виняток **ZeroDivisionError**, запобігли аварійному завершенню решти коду через необроблений виняток.

При роботі з винятками в Python рекомендується включати до складу блоків **try-except** і розділ **else**, і розділ **finally**. Розділ **else** дозволяє програмісту налаштувати дії, що виконуються в тому випадку, якщо під час виконання коду, який захищають від проблем, не було викликано винятків. А розділ **finally** дозволяє забезпечити обов'язкове виконання деяких заключних операцій, на кшталт звільнення ресурсів, незалежно від факту виникнення винятків (ось і ось корисні матеріали про це).

Наприклад – розглянемо ситуацію, коли потрібно прочитати дані з файлу та виконати якісь дії з цими даними. Якщо при читанні файлу виникне виняток, розробник може вирішити, що треба залогувати помилку і зупинити виконання подальших операцій. Але у будь-якому випадку файл потрібно правильно закрити.

Використання розділів **else** і **finally** дозволяє зробити саме так — обробити дані звичайним чином у тому випадку, якщо винятків не виникло, або обробити будь-які винятки, але, як би не розвивалися події, в результаті закрити файл. Без цих розділів код страждав би уразливістю як витоку ресурсів чи неповної обробки помилок. В результаті виявляється, що **else** і **finally** відіграють

найважливішу роль у створенні стійких до помилок та надійних програм.

```
 try:
    file = open("file.txt", "r")
    print("Successful opened the file")
except FileNotFoundError:
    print("File Not Found Error: No such file
or directory")
    exit()
except PermissionError:
    print("Permission Denied Error: Access is
denied")
else:
    content = file.read().decode('utf-8')
    processed_data = process_content(content)
finally:
    file.close()
```

У цьому прикладі ми спочатку намагаємося відкрити файл **file.txt** для читання (у подібній ситуації можна використовувати вираз with, який гарантує правильне автоматичне закриття об'єкта після завершення роботи). Якщо під час виконання операцій файлового вводу/виводу виникають помилки **FileNotFoundError** або **PermissionError**, виконуються відповідні розділи **except**. Тут, заради простоти, ми лише виводимо на екран повідомлення про помилки і виходимо з програми, якщо файл не знайдено.

В іншому випадку, якщо в блоці try винятків не виникло, ми продовжуємо роботу, обробляючи вміст файлу в гілці **else**. І нарешті - виконується "прибирання" – файл закривається незалежно від виникнення винятку. Це забезпечує блок.

Застосовуючи структурований підхід до обробки винятків, що нагадує вищеописаний, можна підтримувати свій код у добре організованому стані та забезпечувати його читабельність. При цьому код буде розрахований на боротьбу з потенційними помилками, які можуть

виникнути під час взаємодії із зовнішніми системами або вхідними даними.

Ось кілька рекомендацій, що стосуються обробки помилок у Python:

- Проектуйте код з розрахунком на можливе виникнення помилок. Заздалегідь плануйте пристрій коду з урахуванням можливих збоїв та проектуйте програми так, щоб вони могли гідно обробляти ці збої. Це означає передбачати можливі прикордонні випадки та реалізовувати відповідні обробники помилок.
- Використовуйте змістовні повідомлення про помилки. Зробіть так, щоб програма виводила б, на екран, або файл журналу, докладні повідомлення про помилки, які допоможуть користувачам зрозуміти - що і чому пішло не так. Намагайтеся не застосовувати узагальнені повідомлення про помилки, на кшталт Error occurred або Something bad happened. Натомість подумайте про зручність користувача та покажіть повідомлення, в якому буде надана порада щодо вирішення проблеми або буде наведено посилання на документацію. Постарайтеся дотриматися балансу між виведенням докладних повідомлень і перевантаженням користувальницького інтерфейсу надмірними даними.
- Мінімізуйте побічні ефекти. Намагайтеся мінімізувати наслідки збійних операцій, ізолюючи проблемні розділи коду за допомогою конструкції try-finally або try з використанням with. Зробіть так, щоб після виконання коду, було воно вдалим чи ні, обов'язково виконували б «очисні» операції.
- Ретельно тестуйте код. Забезпечте коректну поведінку обробників помилок у різних сценаріях використання програми, піддавши код всеосяжному тестуванню.
- Регулярно виконуйте рефакторинг коду. Виконуйте рефакторинг фрагментів коду, схильних до помилок, щоб поліпшити їх надійність і продуктивність. Постарайтеся, щоб ваша кодова база була влаштована

за модульним принципом, щоб її окремі частини слабко залежали б один від одного. Це дозволяє незалежним частинам код самостійно еволюціонувати, не впливаючи на інші його частини.

- Логуйте важливі події. Слідкуйте за цікавими подіями програми, записуючи відомості про них у файл журналу або виводячи в консоль. Це допоможе вам виявляти проблеми на ранніх стадіях їх виникнення, не гаючи часу на тривалий аналіз великої кількості неструктурованих логів.

Модулі та пакети

Як правило, програми на мові Python складаються не з одного, а з багатьох текстових файлів, що містять інструкції. При цьому один із файлів використовується як головний, а всі інші додаткові файли підключаються до нього за допомогою імпорту. Говорячи про файли в такому контексті, ми якраз і маємо на увазі модулі.

Модуль (від англ. module) – це окремий файл із програмним кодом мовою Python, який створюється один раз і далі може бути використаний програмами багаторазово.

Для формування модуля необхідно створити звичайний текстовий файл з розширенням ***py** та записати до нього цільові програмні інструкції. Назва файлу при цьому представлятиме ім'я модуля, а сам модуль після створення стане доступним для використання або як незалежний сценарій, або у вигляді розширення, що підключається до інших модулів, дозволяючи тим самим зв'язувати окремі файли у великі програмні системи. При цьому всі імена, яким буде виконано присвоєння на верхньому рівні модуля (тобто всередині файлу модуля поза функціями, класами тощо), стають атрибутами об'єкта модуля, доступними для використання клієнтами у звичному форматі **mod_name.attr**.

Варто розуміти, що будь-яка програма має певну точку входу. Це своєрідне місце з якого стартує наш скрипт. У мовах попередниках цією точкою служила функція **main** і могла бути лише однієї. У нашому випадку допускається відсутність такої, але це знижує якість коду, роблячи його складним і малопередбачуваним (при імпорті код, що міститься на верхньому рівні, виконується). Для того щоб вказати точку входу (може бути вказана тільки в модулях) використовується спеціальна змінна **__name__**, яка містить найменування поточного модуля або пакета. Якщо поточний модуль знаходиться на верхньому рівні виконання (ми явно передали його на виконання Python), то він називається **__main__** незалежно від назви файлу.

Використання модульного підходу у програмуванні дає низку важливих переваг, оскільки модулі:

- забезпечують багаторазове використання програмного коду в різних програмах за рахунок збереження його у файлах з можливістю повторного завантаження та запуску коду стільки разів, скільки потрібно;
- розбивають простір імен програми на окремі замкнуті пакети, зводячи до мінімуму ймовірність конфлікту імен, оскільки всі імена кожного окремого модуля стають видимими переважно лише після імпортування модуля;
- можуть використовуватися для незалежної розробки компонентів, а також реалізації служб або даних для спільного використання за рахунок можливості подальшого імпортування безліччю клієнтів.

Всі модулі Python можна розділити на чотири основні категорії:

- **вбудовані модулі** (від англ. bilt-in) – є базовими можливостями мови і або імпортуються інтерпретатором автоматично, або вимагають лише простого імпортування без необхідності додаткової установки;

- **стандартна бібліотека** (від англ. standard library) – велика колекція додаткових модулів і пакетів, що мають розширені можливості мови, яка входить безпосередньо до складу дистрибутива Python і вимагає лише простого імпортування її модулів без необхідності додаткової установки;
- **сторонні модулі** (від англ. 3rd party) – понад 90 000 модулів та пакетів, які не входять до дистрибутиву Python, але можуть бути встановлені з **Python Package Index (PyPI)** офіційного сайту за допомогою утиліти pip;
- **модулі користувача** – всі модулі програми, які створюються самими розробниками.

На даний момент для Python написано величезну кількість модулів практично на всі випадки життя. Запам'ятати або хоча б швидко ознайомитися з ними звичайно ж неможливо. Однак це не потрібно, оскільки будь-який модуль використовується лише за потребою. Наприклад, якщо доводиться розробляти додаток, у якому використовуються складні математичні обчислення, достатньо знайти та підключити відповідний математичний модуль із вбудованою підтримкою необхідних операцій.

Для прикладу реалізуємо простий модуль, який повертатиме інформацію:

```
# http_get.modules.http_get
def get_dict():
    return {'status': 200, 'data': 'success'}

if __name__ == '__main__':
    print(get_dict())
```

Далі створимо **main.py** файл, в який імпортуємо наш модуль двома різними способами:

```
# main.py
from
ModulesAndPackages.module_examples.http_get.module
s.http_get import get_dict as absolute
```

```
from http_get.modules.http_get import get_dict as relative
def main():
    print(absolute())
    print(relative())


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

Як і у випадку класів, приховування даних модулів у Python регулюється в основному на рівні взаємозв’язків, а не синтаксичних конструкцій. В основному це стосується випадків, коли потрібно зменшити ймовірність марного забруднення простору імен або випадкової зміни значень змінних при імпорті інструкціями **from package import*** (імпорт усіх доступних модулів пакета) та **from module import*** (імпорт усіх доступних імен модуля). Саме для цих випадків передбачено кілька основних угод для приховування даних:

- Імена змінних, які починаються з одного символу нижнього підкреслення, імпортуватимуться інструкцією from module import * не будуть.
- Якщо на верхньому рівні модуля вказати змінну __all__, надавши їй список рядків з іменами змінних, то інструкція from module import * копіюватиме лише ці імена.
- Якщо на верхньому рівні файлу ініціалізації __init__.py пакета вказати змінну __all__, надавши їй список рядків з іменами модулів, то інструкція from package import імпортує зазначені в ньому модулі, хоча при порожньому файлі ініціалізації інструкція взагалі нічого не робить.

Крім можливості імпортування імен модулів у Python є ще й можливість імпортування імен каталогів. Такий підхід дозволяє ефективно організовувати файли у великих системах, а також у багатьох випадках спрощує

налаштування шляху пошуку модулів і зменшує ймовірність конфлікту імен за наявності декількох файлів програм з однаковими іменами.

Пакет (від англ. package) – це окремий каталог Python, в якому містяться модулі та інші пакети, а також обов'язковий файл **__init.py__**, що відповідає за ініціалізацію пакета.

Імпортувати пакети практично так само легко, як і модулі. Тільки при цьому в інструкції **import** потрібно вказувати шлях до необхідного модуля пакета, використовуючи для поділу звичайні крапки.

Розглянемо наступний приклад імпортування пакету:

```
dir_0/
main.py  # Основний модуль програми
 dir_1/
 __init__.py   .
 file_1.py
  dir_2/  # вкладений в dir_1 каталог пакету
  __init__.py  # Файл ініціалізації для dir_2
  file_2.py
   dir_3/  # вкладений в dir_1 каталог пакету
   __init__.py  # Файл инициализации для dir_2
   file_3.py
```

Як бачимо, для створення структури пакета необхідно створити каталог з підкаталогами і в кожному з них розташувати обов'язковий спеціальний файл ініціалізації **__init.py__**, інструкції якого виконуватимуться щоразу при первинному імпорті пакета або підпакета, а також файли з python-кодом, які представлятимуть собою модулі пакета та його підпакетів (їх інструкції, до речі, теж виконуються при первинному імпорті).

Після того, як пакет буде створено, будь-який з його модулів або окремих імен у них можуть бути підключені за допомогою звичайних інструкцій імпорту **import** або **from**, але з одним важливим доповненням: замість одиночного імені модуля необхідно прописувати повний шлях до нього, починаючи від кореневого каталогу пакету та

відокремлюючи вкладені пакети за допомогою точкової нотації. Так імпортування модуля **file_2** ми використовували інструкцію **import dir_1.dir_2.file_2**, а імпортування єдиного імені **var_1** з модуля **file_1** – інструкцію **from dir_1.file_1 import var_1**.

Слід зазначити, що після імпортування модуля або підпакета всі елементи шляху, такі як кореневий каталог, підкаталоги і сам модуль, автоматично стають ланцюжком об'єктів, вкладених один в одного, у вигляді значень відповідних атрибутів. Завдяки цьому всім об'єктам ланцюжка можна звернутися звичайним для атрибутів способом, тобто. за допомогою точкової нотації. Так, після імпорту **import dir_1.dir_2.file_2** ми змогли отримати доступ як до пакету **dir_1**, так і підпакету **dir_1.dir_2**, що дозволило нам вивести їх рядкові уявлення на екран.

Оскільки шлях до модуля часом може бути досить довгим, дозволяється використовувати звичне нам розширення **as**. У нашому прикладі ми скоротили довге ім'я в інструкції import **dir_1.dir_3.file_3 as fl_3**, скориставшись псевдонімом **fl_3** і замінивши весь можливий ланцюжок вкладених об'єктів єдиним об'єктом, пов'язаним з пседнонімом (як наслідок, звернутися до імені **dir_1**, наприклад, вже не звернутися до імені **dir_1**). Так само ми могли б використовувати псевдонім і в інструкції from, прописавши, наприклад, **from dir_1.file_1 import var_1 as v_1**.

Що стосується пакетів і підпакетів, то їх дозволяється імпортувати так само, як і звичайні модулі. При цьому варто мати на увазі, що в процесі імпорту всі інструкції файлів ініціалізації **__init.py__** виконуються в порядку вкладеності пакетів, а їх глобальні змінні автоматично стають доступними в цільовому модулі як атрибути імпортованих імен

Документування коду

Документування коду в Python – досить важливий аспект, адже від цього часом залежить читання та швидкість розуміння вашого коду, як іншими людьми, так і вами через півроку.

PEP 257 описує угоди, пов'язані з рядками документації python, розповідає про те, як потрібно документувати код.

Мета цього PEP – стандартизувати структуру рядків документації: що вони повинні включати, і як це написати (не торкаючись питання синтаксису рядків документації). Цей PEP визначає угоди, а не правила або синтаксис.

При порушенні цих угод найгірше, чого можна очікувати - деяких несхвальних поглядів. Але деякі програми (наприклад, **docutils**) використовують домовленості, тому слідування їм дасть вам найкращі результати.

Рядки документації - рядкові літерали, які є першим оператором у модулі, функції, класі чи визначенні методу. Такий рядок документації стає спеціальним атрибутом цього об'єкта.

Усі модулі повинні, як правило, мати рядки документації, і всі функції та класи, що експортуються модулем, також повинні мати рядки документації. Публічні методи (у тому числі **__init__**) також мають мати рядки документації. Пакет модулів може бути документований у **__init__.py**.

Для узгодженості завжди використовуйте **"""triple double quotes"""** для рядків документації. Використовуйте **r"""raw triple double quotes"""**, якщо ви будете використовувати зворотну косу в рядку документації.

Існує дві форми рядків документації: однорядкова та багаторядкова.

Однорядкова призначена для справді очевидних випадків. Вони повинні вміщатися на одному рядку. Наприклад:

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
```

Використовуйте потрійні лапки, навіть якщо документація міститься на одному рядку. Потім буде простіше доповнити її.

Закривають лапки на тому ж рядку. Це виглядає краще. Не повинно бути порожніх рядків перед документацією або після неї.

Багаторядкова документація складаються з однорядкового рядка документації з наступним порожнім рядком, а потім докладнішим описом. Перший рядок може бути використаний автоматичними засобами індексації, тому важливо, щоб він знаходився на одному рядку і був відокремлений від решти документації порожнім рядком. Перший рядок може бути на тому ж рядку, де і лапки, що відкривають, або на наступному рядку. Вся документація повинна мати такий самий відступ, як лапки на першому рядку (див. приклад нижче).

Рядки документації скрипта (самостійної програми) мають бути доступні як "повідомлення про використання", надрукованій, коли програма викликається з некоректними або відсутніми аргументами (або, можливо, з опцією "-h", для допомоги). Такий рядок документації повинен документувати функції програми та синтаксис командного рядка, змінні оточення та файли. Повідомлення використання може бути досить складним (кілька екранів) і має бути достатнім для нового користувача для використання програми належним чином, а також повний довідник з усіма варіантами та аргументами для досвідченого користувача.

Тема 7. Процеси. Потоки. Підпрограми. Паралелізм. Конкурування. Синхронізація.

Багатопоточність (concurrency) - це можливість виконання багатьох завдань (потоків) паралельно в одному програмному середовищі. Багатопоточність використовується для вирішення таких завдань:

1. Виконання завдань паралельно: В багатьох випадках, використання потоків може значно покращити продуктивність програм, дозволяючи виконувати декілька завдань одночасно. Наприклад, при обробці великої кількості даних, багатопоточність дозволяє розділити завдання на менші частини і виконувати їх паралельно.
2. Відгук користувача: Багатопоточність дозволяє створювати інтерактивні програми, які можуть відповідати на дії користувача під час виконання інших завдань. Це важливо для розробки інтерфейсів користувача та мережевих додатків.
3. Забезпечення обробки подій: В багатьох програмах потрібно обробляти події, які виникають асинхронно, такі як натискання клавіш, введення миші або приходження повідомлень. Використання потоків дозволяє реагувати на ці події без блокування інших операцій.

Проте багатопоточність може призвести до різних проблем:

1. Гонки за ресурсами (Race Conditions): Коли декілька потоків спробують одночасно отримати доступ до спільних ресурсів, можуть виникати конфлікти, що призводять до непередбачуваної поведінки.

2. Блокування (Deadlocks): Потоки можуть увійти в стан блокування, коли вони очікують одне на одного або на доступ до ресурсів. Це призводить до зупинки виконання програми.
3. Гонки за даними (Data Races): Якщо декілька потоків одночасно змінюють спільні дані без синхронізації, можуть виникнути помилки в роботі програми.
4. Споживач-виробник (Producer-Consumer) проблеми: В багатьох випадках потоки виробників виробляють дані, а потоки споживачів споживають їх. У випадку недостатньої синхронізації ця взаємодія може викликати проблеми.

Розглянемо більше деталей про кожен з аспектів: процеси, потоки, підпрограми, паралелізм, конкурування та синхронізацію в програмуванні:

1. **Процеси (Processes)**:
    - Процес - це окремий екземпляр програми, що виконується на комп'ютері. Кожен процес має власний об'єм пам'яті і виконується паралельно з іншими процесами. Вони ізольовані один від одного.
    - У Python ви можете створювати та керувати процесами за допомогою модуля `multiprocessing`.
    - Процеси корисні, коли потрібно виконувати завдання, які можна розділити на паралельні процеси.
2. **Потоки (Threads)**:
    - Поток - це легкий підпроцес, який виконується в межах процесу. Він ділить ресурси процесу, але має власний стек викликів.

- У Python для створення та керування потоками використовується модуль `threading`.
- Потоки корисні для задач, які можна виконувати одночасно і вимагають спільного доступу до пам'яті.

3. **Паралелізм (Parallelism)**:
   - Паралелізм - це концепція виконання багатьох завдань одночасно для покращення продуктивності.
   - В програмуванні це досягається за допомогою процесів або потоків.
   - Паралельне виконання дозволяє програмі працювати швидше та ефективніше, використовуючи доступні ресурси комп'ютера.
4. **Конкурування (Concurrency)**:
   - Конкурування відбувається, коли багато задач виконуються одночасно, але не обов'язково одночасно.
   - Це може відбуватися в одному потоці чи в одному процесі.
   - Конкуруючи завдання мають доступ до спільних ресурсів та повинні вирішувати проблеми з одночасним доступом до них.
5. **Синхронізація (Synchronization)**:
   - Синхронізація - це механізм, який використовується для забезпечення правильної взаємодії між конкуруючими завданнями або потоками.
   - Вона може включати блокування ресурсів, використання семафорів або інших засобів, що дозволяють встановити порядок виконання.

Важливо розуміти, коли і де використовувати кожен з цих аспектів для створення надійних і ефективних програм. У Python ви маєте доступ до широкого спектру інструментів та бібліотек для роботи з цими аспектами багатозадачності.

У Python є кілька способів реалізації цих аспектів.

1. **Процеси**: Процес - це окремий запущений екземпляр програми, який має свій власний об'єм пам'яті та виконується паралельно з іншими процесами. В Python ви можете створювати процеси за допомогою модуля `multiprocessing`.

```
import multiprocessing

def worker_function():
    print("This is a worker process.")

if __name__ == '__main__':
    process = 
multiprocessing.Process(target=worker_function)
    process.start()
    process.join()
```

2. **Потоки**: Поток - це легший аналог процесу, який виконується в межах одного процесу. В Python ви можете створювати потоки за допомогою модуля `threading`.

```
import threading

def worker_function():
    print("This is a worker thread.")

thread = threading.Thread(target=worker_function)
thread.start()
thread.join()
```

3. **Паралелізм**: Паралелізм - це виконання декількох завдань одночасно для покращення продуктивності. У Python ви можете використовувати бібліотеки, такі як `concurrent.futures` або `asyncio` для паралельного виконання завдань.

```
import concurrent.futures

def some_task(task_name):
    print(f"Starting task {task_name}")
    # Деяка обробка або виконання завдання
    print(f"Completed task {task_name}")

if __name__ == "__main__":
    task_names = ["Task1", "Task2", "Task3"]

    # Використовуємо ThreadPoolExecutor для
створення потоків
    with concurrent.futures.ThreadPoolExecutor()
as executor:
        # Запускаємо кожне завдання в окремому
потоці
        executor.map(some_task, task_names)
```

Цей код створює три потоки, які виконують завдання `some_task` паралельно. Ви можете використовувати `ThreadPoolExecutor` для створення потоків або `ProcessPoolExecutor`, щоб створити процеси для паралельного виконання завдань.

4. **Конкурування (Concurrency)**: Конкурування означає виконання багатьох завдань паралельно, але не обов'язково одночасно. У Python, ви можете використовувати потоки або асинхронний код для конкуруючого виконання.

```
import threading

def print_numbers():
    for i in range(1, 6):
```

```
        print(f"Number: {i}")

def print_letters():
    for letter in 'abcde':
        print(f"Letter: {letter}")

if __name__ == "__main__":
    # Створюємо два потоки для виконання функцій
    t1 = threading.Thread(target=print_numbers)
    t2 = threading.Thread(target=print_letters)

    # Запускаємо потоки
    t1.start()
    t2.start()

    # Очікуємо завершення обох потоків
    t1.join()
    t2.join()

    print("Both threads have finished")
```

У цьому прикладі ми створюємо два потоки, один для виведення чисел від 1 до 5, інший - для виведення літер від 'a' до 'e'. Обидва потоки запускаються паралельно, і ми використовуємо `join`, щоб зачекати, поки обидва потоки завершаться.

5. **Синхронізація**: Синхронізація важлива для уникнення конфліктів та гарантії правильної обробки одночасних операцій. В Python, ви можете використовувати механізми блокування, такі як `Lock`, для синхронізації доступу до ресурсів з різних потоків або процесів.

```
import threading

counter = 0
lock = threading.Lock()

def increment_counter():
    global counter
```

```
    with lock:
        counter += 1

threads = []
for _ in range(10):
    thread =
threading.Thread(target=increment_counter)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

print("Counter:", counter)
```

Ці аспекти важливі для створення ефективних та надійних програм, особливо в великих проектах та серверних застосунках. Вони допомагають досягти одночасності та підвищити продуктивність ваших програм.

Global Interpreter Lock

GIL, або Global Interpreter Lock, це механізм, який використовується в реалізаціях CPython (це стандартна реалізація мови Python) з метою забезпечення безпечності при виконанні коду на Python. GIL дозволяє виконувати лише одну операцію одночасно, навіть на мультипроцесорних системах.

GIL створений для уникнення конфліктів при доступі до об'єктів Python з різних потоків. Він гарантує, що лише один потік може виконувати Python-код у будь-який момент. Це робить важкою паралельну обробку в багатьох потоках на рівні CPU, оскільки лише один потік може активно виконувати код Python.

Хоча GIL забезпечує безпеку виконання коду Python та уникає багатьох проблем з паралельністю, він також обмежує потенційну продуктивність програм, які

використовують багато потоків. В ситуаціях, коли задачі можуть бути паралельно виконані на різних процесорах, GIL може стати перешкодою.

GIL не є проблемою для всіх типів програм, і в багатьох випадках він навіть корисний. Але для програм, які активно використовують багато потоків та мають обчислювально інтенсивні завдання, GIL може стати обмеженням продуктивності. У таких випадках розглядають альтернативні реалізації Python, такі як Jython або IronPython, які не використовують GIL, або використовують інші мови програмування, які не мають цього обмеження.

Розглянемо на прикладі проблему GIL (Global Interpreter Lock) та його вплив на паралельність в мові програмування Python за допомогою прикладу.

Уявімо, що у нас є завдання, яке потрібно обчислити для кожного елемента великого списку. Ми можемо спробувати реалізувати це обчислення за допомогою потоків для розділення роботи між різними потоками. Однак через GIL, це може не працювати так, як очікується. Ось приклад:

```
import threading

def обчислити_завдання(index):
    global total
    for _ in range(1000000):
        total += 1

total = 0
threads = []

for i in range(10):
    thread = 
threading.Thread(target=обчислити_завдання, 
args=(i,))
```

```
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

print("Загальна сума:", total)
```

У цьому прикладі ми створюємо 10 потоків, кожен з яких виконує `обчислити_завдання`, який просто додає до змінної `total`. Очікується, що після завершення усіх потоків `total` буде рівним 10 мільйонам.

Але через GIL, який не дозволяє виконувати багато операцій одночасно в одному процесі Python, цей приклад може не дати очікуваний результат. Замість 10 мільйонів, ми можемо отримати значно менше.

Щоб уникнути цієї проблеми, можна розглянути використання процесів замість потоків для паралельних обчислень. Однак важливо пам'ятати, що обмін даними між процесами може бути складнішим, ніж між потоками, і потребує використання інструментів, таких як черги чи спільний ресурс для зберігання даних між процесами.

Thread-Based Concurrency

Thread-Based Concurrency - це один із способів досягнення конкурентності в мові програмування Python, використовуючи потоки (threads). Потоки дозволяють виконувати різні частини програми паралельно для поліпшення продуктивності. Нижче наведено огляд методів та прикладів, які використовуються для роботи з потоками в Python:

Створення потоків: Ви можете створити поток за допомогою модуля `threading` в Python. Ось приклад створення потоку:

```
import threading

def worker():
    print("Робочий потік")

thread = threading.Thread(target=worker)
thread.start()
```

Завершення потоків**:** Ви можете чекати на завершення потоку за допомогою методу `.join()`. Це допомагає забезпечити, що основний потік буде чекати, доки інші потоки не завершаться:

```
thread = threading.Thread(target=worker)
thread.start()
thread.join()
```

Передача аргументів до потоків**:** Ви можете передавати аргументи до функції, яку виконує потік, як параметри при створенні потоку:

```
def worker_with_args(name):
    print(f"Робочий потік з ім'ям {name}")

thread = threading.Thread(target=worker_with_args,
args=("Потік 1",))
thread.start()
```

Багато потоків**:** Ви можете створити багато потоків для виконання однієї функції. У цьому випадку вони будуть виконувати функцію паралельно:

```
threads = []
for i in range(5):
    thread = threading.Thread(target=worker)
    threads.append(thread)
```

```
    thread.start()

for thread in threads:
    thread.join()
```

Захист від гонки: Оскільки потоки можуть спільно використовувати ресурси, існує ризик гонок. Для захисту від цього можна використовувати м'ютекси (mutex) з модуля `threading`, які блокують доступ до ресурсів одного потоку іншим. Наприклад:

```
shared_variable = 0
lock = threading.Lock()

def update_shared_variable():
    global shared_variable
    with lock:
        shared_variable += 1

threads = []
for _ in range(5):
    thread = 
threading.Thread(target=update_shared_variable)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

print(shared_variable)  # Результат буде 5
```

Завершення потоку: Якщо ви бажаєте завершити виконання потоку, ви можете використовувати флаги або внутрішні засоби для припинення виконання потоку безпечним способом.

Таким чином, Thread-Based Concurrency дозволяє створювати багато потоків для виконання різних функцій паралельно і поліпшує продуктивність вашої програми. Однак важливо бути обережним і уникати гонок, які

можуть виникнути при одночасному доступі до спільних ресурсів.

Методами синхронізації

Методи синхронізації в мові програмування Python потрібні для керування одночасним доступом декількох потоків або процесів до спільних ресурсів чи даних. Ось більш детальне розглядання, для чого вони потрібні:

1. Уникнення гонок (Race Conditions): Гонка виникає, коли два або більше потоки чи процеси намагаються одночасно змінити спільний ресурс, що може призвести до непередбачуваних результатів. Синхронізація допомагає уникнути гонок і забезпечує правильний порядок доступу до ресурсу.
2. Забезпечення Консистентності Даних: Під час одночасного доступу до даних важливо забезпечити, щоб дані були консистентними. Синхронізація допомагає забезпечити правильну консистентність даних, навіть під час паралельного виконання.
3. Поділ Ресурсів: Якщо потоки чи процеси мають спільний доступ до обмежених ресурсів, таких як файловий дескриптор, сокет, база даних, то синхронізація допомагає управляти цими ресурсами та поділити їх між потоками.
4. Зменшення Витрати Ресурсів: Паралельні операції можуть вимагати багато ресурсів, таких як пам'ять чи обчислювальна потужність. Синхронізація дозволяє керувати цими ресурсами та ефективно використовувати їх.
5. Забезпечення Справедливості: Синхронізація допомагає забезпечити справедливий доступ до

ресурсів для різних потоків чи процесів, дотримуючись певних правил чи пріоритетів.

6. Уникнення Виснаження Ресурсів: Синхронізація допомагає уникнути виснаження ресурсів, коли деякі потоки чекають на доступ до ресурсу, блокуючи його надмірним часом.

Засоби синхронізації в Python включають у себе замки, семафори, умовні замки, події, бар'єри та інші. Вибір конкретного методу залежить від потреб вашого застосунку, та важливо збалансувати ефективність та правильність роботи програми.

Розглянемо приклади з різними методами синхронізації в потоках Python:

Lock: Використовуючи замки, ви можете захоплювати та звільняти доступ до критичних ресурсів. Це гарантує, що тільки один потік може виконувати код, захопленням замка.

```
import threading

shared_resource = 0
lock = threading.Lock()

def worker():
    global shared_resource
    with lock:  # Захоплюємо замок
        shared_resource += 1
        print(f"Загальний ресурс:
{shared_resource}")

threads = []
for _ in range(4):
    thread = threading.Thread(target=worker)
    threads.append(thread)
    thread.start()
```

```
for thread in threads:
    thread.join()
```

Lock (без захоплення): Іноді синхронізація може бути зайвою. Якщо ви не хочете блокувати потоки, але просто хочете отримати доступ до об'єкта з безпекою від багатопоточності, ви можете використовувати простий метод без замку або інших об'єктів синхронізації.

```
from threading import Lock

lock = Lock()

def safe_access():
    with lock:
        print("Доступ забезпечено")

safe_access()
```

Умовні змінні (Condition Variables): Умовні змінні використовуються для забезпечення взаємодії між потоками. Один потік може чекати на подію, яку сповіщає інший потік.

```
import threading

data = None
condition = threading.Condition()

def producer():
    global data
    with condition:
        data = "Дані від виробника"
        condition.notify()  # Повідомляємо інші
потоки

def consumer():
    global data
    with condition:
        condition.wait()  # Очікуємо на подію
```

```
        print(f"Споживач отримав: {data}")

threads = [threading.Thread(target=producer),
threading.Thread(target=consumer)]
for thread in threads:
    thread.start()

for thread in threads:
    thread.join()
```

Семафори (Semaphores): Семафори використовуються для обмеження доступу до певної кількості ресурсів. Вони можуть вказувати, скільки потоків може одночасно користуватися ресурсами.

```
import threading

semaphore = threading.Semaphore(2)  # Два потоки
можуть одночасно користуватися ресурсами

def worker():
    with semaphore:  # Захоплюємо семафор
        print(f"Потік отримав доступ")
        # Виконуємо роботу з ресурсом

threads = []
for _ in range(4):
    thread = threading.Thread(target=worker)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

Це всього лише декілька методів синхронізації, які доступні в Python для керування потоками. Вибір методу залежить від конкретної задачі та потреб вашої програми.

Event (Подія): Події дозволяють одному або більше потоків чекати на виникнення певної події. Це корисно,

коли потоки мають виконувати роботу, якщо виникає певна умова.

```
import threading

event = threading.Event()

def worker():
    print("Потік чекає на подію")
    event.wait()  # Чекаємо на виникнення події
    print("Подія відбулася")

threads = [threading.Thread(target=worker),
threading.Thread(target=worker)]
for thread in threads:
    thread.start()

# Викликаємо подію через певний час
import time
time.sleep(2)
event.set()  # Викликаємо подію

for thread in threads:
    thread.join()
```

Barrier (Бар'єр): Бар'єри використовуються для синхронізації багатьох потоків, які чекають один на одного перед виконанням певної дії.

```
import threading

barrier = threading.Barrier(3)  # Встановлюємо
бар'єр для 3 потоків

def worker():
    print(f"Потік прибув до бар'єру")
    barrier.wait()  # Чекаємо всі інші потоки
    print(f"Потік перейшов бар'єр")

threads = [threading.Thread(target=worker) for _
in range(3)]
for thread in threads:
```

```
    thread.start()

for thread in threads:
    thread.join()
```

Вибір методу залежить від конкретної задачі та потреб вашого застосунку.

Тема 8. ГРАФІЧНІ КОРИСТУВАЦЬКІ ІНТЕРФЕЙСИ: БІБЛІОТЕКА PYQT.

PyQt — це бібліотека Python для створення програм із графічним інтерфейсом за допомогою інструментарію **Qt**. Створена в Riverbank Computing, PyQt є вільним програмним забезпеченням (за ліцензією GPL) і розробляється з 1999 року. Остання версія PyQt6 – на основі Qt 6 – випущена у 2021 році, і бібліотека продовжує оновлюватись. Цей посібник також можна використовувати для PySide2, PySide6 і PyQt5.

На сьогодні використовуються дві основні версії: PyQt5 на основі Qt5 та PyQt6 на основі Qt6. Обидві майже повністю сумісні, за винятком імпорту та відсутності підтримки деяких сучасних модулів з Qt6. У PyQt6 вносяться зміни у роботу просторів імен та прапорів, але ними легко управляти. Далі дізнаємося, як використовувати PyQt6 для створення настільних програм.

Спочатку створимо кілька простих вікон на робочому столі, щоб переконатися, що PyQt працює і розберемо базові поняття. Потім коротко вивчимо цикл подій і те, як він пов'язаний із програмуванням графічного інтерфейсу на Python.

Установка PyQt

Для встановлення бібліотеки необхідно виконати наступну команду:

```
pip install pyqt6
pip install pyqt-tools
```

Спочатку створимо новий файл Python з будь-якою назвою (наприклад, app.py) і збережемо його. Вихідний код програми показано нижче:

```
from PyQt6.QtWidgets import QApplication, QWidget
import sys


app = QApplication(sys.argv)
```

```
# створюємо віджет виджет Qt — вікно
window = QWidget()
window.show()  # важливо: вікно приховано за
замовчуванням
# запускаєм цикл подій
app.exec()
```

Запускаємо програму з командного рядка, як і будь-який скрипт Python:

```
python3 app.py
```

Виконавши його, побачимо вікно. У Qt автоматично створюється вікно із звичайним оформленням, можливістю його перетягувати та змінювати розмір. Те, що ви побачите, залежить від платформи, де цей приклад виконується. Ось як відображається це вікно на Windows, macOS та Linux (Ubuntu):

Розберемо код порядково, щоб зрозуміти, що саме відбувається. Спочатку ми імпортуємо класи PyQt для програми: тут це обробник програми QApplication і базовий порожній віджет графічного інтерфейсу QWidget (обидва з модуля QtWidgets):

```
from PyQt6.QtWidgets import QApplication, QWidget
```

Можливий ще **from import** *, але цей вид імпорту зазвичай не вітається в Python. Далі створюємо екземпляр **QApplication** і передаємо **sys.arg** (список Python з аргументами командного рядка, що передаються додатком):

```
app = QApplication(sys.argv)
```

Якщо не будете використовувати аргументи командного рядка для керування Qt, передайте порожній список:

```
app = QApplication([])
```

Потім створюємо екземпляр **QWidget**, використовуючи ім'я змінної window:

```
window = QWidget()
window.show()
```

У Qt всі віджети верхнього рівня – вікна, тобто вони не мають батьківського елемента і вони не вкладені в інший віджет або макет. В принципі, вікно можна створити, використовуючи будь-який віджет.

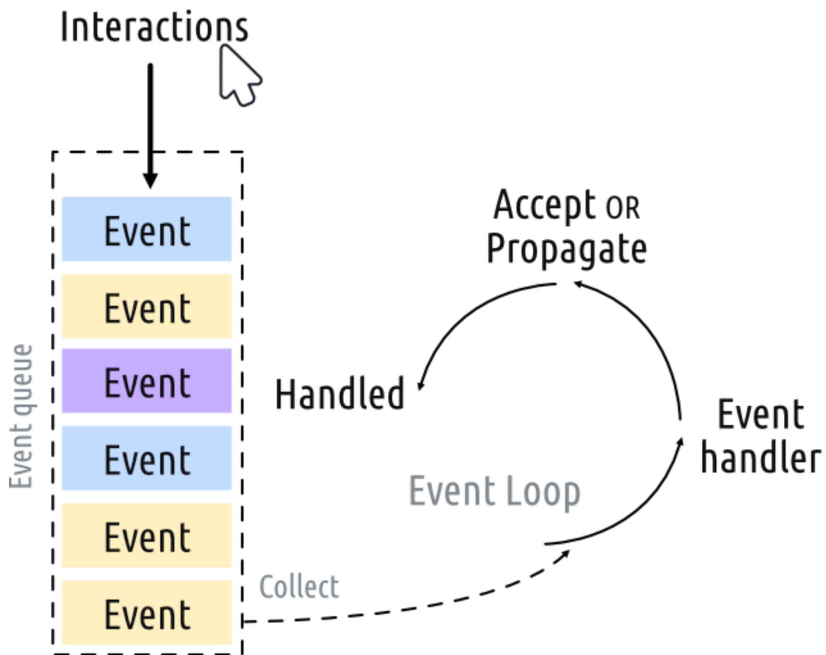
Віджети без батьківського елемента за замовчуванням невидимі. Тому після створення об'єкта window необхідно завжди викликати **.show()**, щоб зробити його видимим. **.show()** можна видалити, але тоді, запустивши програму, ви не зможете вийти з неї.

Нарешті викликаємо **app.exec()**, щоб запустити цикл події.

Цикл подій

Перш ніж вивести вікно на екран, розберемо ключові поняття щодо організації додатків у світі Qt.

Основний елемент усіх програм у Qt - клас **QApplication**. Для роботи кожному додатку потрібен один і лише один об'єкт QApplication, який містить цикл подій програми. Це основний цикл, що управляє всією взаємодією користувача з графічним інтерфейсом:



При кожній взаємодії з застосунком – чи то натискання клавіші, клацання або рух миші – генерується подія, яка міститься в чергу подій. У циклі подій черга перевіряється на кожній ітерації: якщо знайдено подію, що очікує, вона разом з управлінням передається певному обробнику цієї події. Останній обробляє його, потім повертає управління в цикл подій і чекає на нові події. Для кожної програми виконується лише один цикл подій.

Клас **QApplication** містить цикл подій Qt (потрібний один екземпляр QApplication). Додаток чекає у циклі подій нову подію, яка буде згенерована під час виконання дії. Завжди виконується лише один цикл подій.

QMainWindow

Отже, Qt будь-які віджети можуть бути вікнами. Наприклад, якщо замінити **QtWidget** на **QPushButton**. У цьому прикладі виходить вікно з однією кнопкою:

```
import sys
from PyQt6.QtWidgets import QApplication, QPushButton

app = QApplication(sys.argv)

window = QPushButton("Push Me")
window.show()
app.exec()
```

Класно, але не дуже корисно насправді: рідко коли потрібен інтерфейс користувача, що складається тільки з одного елемента управління. Зате можливість за допомогою макетів вкладати одні віджети в інші дозволяє створювати складні інтерфейси користувача всередині порожнього **QWidget**.

У Qt вже є рішення для вікна – віджет **QMainWindow**, що має стандартні функції вікна для використання в додатках, що містить панелі інструментів, меню, рядок стану, віджети, що закріплюються, і багато іншого. Розглянемо ці розширені функції пізніше, а поки додамо додаток простий, порожній QMainWindow:

```
import sys
from PyQt6.QtWidgets import QApplication, QMainWindow

app = QApplication(sys.argv)

window = QMainWindow()
window.show()
app.exec()
```

Запускаємо та бачимо головне вікно.

Додамо контент. Щоб зробити вікно, що налаштовується, краще створити підклас **QMainWindow**, а потім налаштувати вікно в блоці **__init__**. Так вікно стане незалежним щодо поведінки. Отже, додаємо підклас **QMainWindow - MainWindow**:

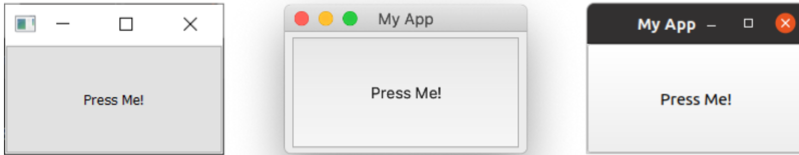
```
import sys

from PyQt6.QtCore import QSize, Qt
from PyQt6.QtWidgets import QApplication, QMainWindow,
QPushButton

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")
        button = QPushButton("Press Me!")
        self.setCentralWidget(button)


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

Для цього демо використовуємо **QPushButton**. Основні віджети Qt завжди імпортуються з простору імен QtWidgets, як і класи QMainWindow та QApplication. При використанні **QMainWindow** використовуємо **.setCentralWidget** для розміщення віджету (тут віджет - QPushButton) в QMainWindow, за замовчуванням він займає все вікно. Як додавати у вікна кілька віджетів? Про це поговоримо розглянемо у посібнику з макетів.

При створенні підкласу із класу Qt, щоб дозволити Qt налаштовувати об'єкт, завжди потрібно викликати функцію super **__init__**.

У блоці **__init__** спочатку використовуємо **.setWindowTitle()**, щоб змінити заголовок головного вікна. Потім додаємо перший віджет QPushButton в середину вікна. Це один із основних віджетів Qt. Під час створення кнопки можна ввести текст, який відображатиметься на ній. Викликаємо **.setCentralWidget()** у вікні. Це спеціальна функція QMainWindow, яка дозволяє встановити віджет на середину вікна.

Запускаємо та знову бачимо вікно, але цього разу з віджетом **QPushButton** у центрі:



Слоти та сигнали

Раніше ми розглянули класи QApplication та QMainWindow, цикл подій та додали у вікно простий віджет. А тепер вивчимо механізми Qt для взаємодії віджетів та вікон один з одним. До цієї статті внесено зміни, пов'язані з PyQt6.

Ми створили вікно і додали до нього простий віджет push button, але кнопка поки що неактивна. Потрібно зв'язати дію натискання кнопки з тим, що відбувається. У Qt це робиться за допомогою сигналів та слотів чи подій.

Сигнали – це повідомлення, що надсилаються віджетами, коли щось відбувається. Цим «чимось» може бути будь-що – натискання кнопки, зміна тексту в полі введення або зміна тексту у вікні. Багато сигналів ініціюються у відповідь на дії користувача, але не тільки: у сигналах можуть надсилатися дані з додатковим контекстом.

Слоти Qt — це приймачі сигналів. Слотом у додатку на Python можна зробити будь-яку функцію (або метод), просто підключивши сигнал. Функція, що приймає, отримує дані, що надсилаються їй у сигналі. У багатьох віджетів Qt є вбудовані слоти, а значить віджети можна підключати один до одного безпосередньо.

Розглянемо основні сигнали Qt та їх використання для підключення віджетів у додатках. Збережіть цей код програми у файлі app.py:

```
import sys
from PyQt6.QtWidgets import QApplication, QMainWindow,
QPushButton

class MainWindow(QMainWindow):
    def __init__(self):
        super(MainWindow, self).__init__()
        self.setWindowTitle("My App")

app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

Сигнали QPushButton

Зараз ми маємо QMainWindow із центральним віджетом QPushButton. Підключимо цю кнопку до методу Python. Створимо простий налаштований слот **the_button_was_clicked**, який приймає сигнал clicked від QPushButton:

```
import sys
from PyQt6.QtWidgets import QApplication, QMainWindow,
QPushButton


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("My App")
        button = QPushButton("Press Me!")
        button.setCheckable(True)

button.clicked.connect(self.the_button_was_clicked)
        self.setCentralWidget(button)

def the_button_was_clicked(self):
        print("Clicked!")
```

```
app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

Запускаємо. Якщо натиснути кнопку, у консолі з'явиться текст Clicked! («Натиснута!»):

```
Clicked!
Clicked!
Clicked!
Clicked!
```

За допомогою сигнала повідомляється про натиснутий стан кнопки. Для звичайних кнопок це значення завжди є False, тому перший слот проігнорував ці дані. Увімкнемо можливість натискання кнопки, щоб побачити цей ефект. Нижче додається другий слот та виводиться стан натискання:

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("My App")
        button = QPushButton("Press Me!")
        button.setCheckable(True)

button.clicked.connect(self.the_button_was_clicked)

button.clicked.connect(self.the_button_was_toggled)

        self.setCentralWidget(button)

    def the_button_was_clicked(self):
        print("Clicked!")

    def the_button_was_toggled(self, checked):
        print("Checked?", checked)
```

Якщо натиснути кнопку, вона підсвітиться і стане checked («Натиснутою»). Щоб вимкнути її, натискаємо ще раз:

```
Clicked!
Checked? True
Clicked!
Checked? False
Clicked!
Checked? True
Clicked!
Checked? False
```

До сигналу підключається скільки завгодно слотів, у яких можна реагувати відразу кілька версій сигналів.

Збереження даних

Поточний стан віджету на Python часто зберігається в змінній, що дозволяє працювати зі значеннями без доступу до вихідного віджету. Причому їх зберігання використовуються окремі змінні чи словник. У наступному прикладі зберігаємо значення кнопки checked («Натиснута») у змінній **button_is_checked** у **self**:

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.button_is_checked = True
        self.setWindowTitle("My App")

        button = QPushButton("Press Me!")
        button.setCheckable(True)

button.clicked.connect(self.the_button_was_toggled)
        button.setChecked(self.button_is_checked)
        self.setCentralWidget(button)

    def the_button_was_toggled(self, checked):
        self.button_is_checked = checked

        print(self.button_is_checked)
```

Спочатку встановлюємо змінну значення за промовчанням True, а потім використовуємо це значення, щоб встановити вихідний стан віджету. Коли стан віджета змінюється, отримуємо сигнал і оновлюємо змінну.

Ця ж схема застосовна до будь-яких віджетів PyQt. Якщо у віджеті немає сигналу, яким надсилається поточний стан, потрібно отримати значення віджету прямо в обробнику. Наприклад, тут ми перевіряємо стан checked («Натиснута») у натиснутому обробнику:

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.button_is_checked = True

        self.setWindowTitle("My App")

        self.button = QPushButton("Press Me!")
        self.button.setCheckable(True)

self.button.released.connect(self.the_button_was_releas
ed)
        self.button.setChecked(self.button_is_checked)

        self.setCentralWidget(self.button)

    def the_button_was_released(self):
        self.button_is_checked =
self.button.isChecked()

        print(self.button_is_checked)
```

Збережемо посилання на кнопку в self, щоб отримати доступ до неї в слоті.

Сигнал **released** спрацьовує, коли кнопка відпускається, при цьому стан натискання не надсилається. Його одержують із кнопки в обробнику, використовуючи **.isChecked()**.

Зміна інтерфейсу

Обновимо метод слота, щоб змінити кнопку, змінивши текст, відключивши її і зробивши її недоступною. І відключимо поки що стан, що допускає натискання:

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")
        self.button = QPushButton("Press Me!")

self.button.clicked.connect(self.the_button_was_clicked
)
        self.setCentralWidget(self.button)

    def the_button_was_clicked(self):
        self.button.setText("You already clicked me.")
        self.button.setEnabled(False)
        self.setWindowTitle("My Oneshot App")
```

Знову потрібний доступ до кнопки в методі **the_button_was_clicked**, тому зберігаємо посилання на неї в **self**. Щоб змінити текст кнопки, передаємо str в .setText(). Щоб вимкнути кнопку, викликаємо **.setEnabled()** із аргументом False. І запускаємо програму. Якщо натиснути кнопку, текст зміниться і кнопка стане недоступною.

У методах слота можна не тільки змінювати кнопку, яка активує сигнал, але й робити все, що завгодно. Наприклад, змінити заголовок вікна, додавши в метод **the_button_was_clicked** цей рядок:

```
self.setWindowTitle("A new window title")
```

Більшість віджетів, у тому числі QMainWindow, мають сигнали. У наступному, більш складному прикладі підключимо сигнал **.windowTitleChanged** QMainWindow до користувальницькому методу слота. А також зробимо для цього слота новий заголовок вікна:

```
from PyQt6.QtWidgets import QApplication, QMainWindow,
QPushButton

import sys
from random import choice
window_titles = [
    'My App','My App','Still My App','Still My
App','What on earth','What on earth','This is
surprising','This is surprising','Something went
wrong']
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.n_times_clicked = 0
        self.setWindowTitle("My App")
        self.button = QPushButton("Press Me!")

self.button.clicked.connect(self.the_button_was_clicked
)
self.windowTitleChanged.connect(self.the_window_title_c
hanged)
        self.setCentralWidget(self.button)

    def the_button_was_clicked(self):
        print("Clicked.")
        new_window_title = choice(window_titles)
        print("Setting title:  %s" % new_window_title)
        self.setWindowTitle(new_window_title)

    def the_window_title_changed(self, window_title):
        print("Window title changed: %s" %
window_title)

        if window_title == 'Something went wrong':
            self.button.setDisabled(True)
app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

Спочатку створюємо список заголовків вікна і вибираємо один із них навмання, використовуючи

вбудовану функцію Python **random.choice**(). Підключаємо метод слота **the_window_title_changed** до сигналу вікна **.windowTitleChanged**.

При натисканні на кнопку заголовок вікна випадково зміниться. Якщо новий заголовок вікна зміниться на Something went wrong («Щось пішло не так»), кнопка відключиться.

Натискайте кнопку, поки заголовок не зміниться на Something went wrong. У цьому прикладі варто звернути увагу ось на що:

Сигнал **windowTitleChanged** під час встановлення заголовка вікна видається не завжди. Він спрацьовує тільки якщо новий заголовок відрізняється від попереднього: якщо один і той же заголовок встановлюється кілька разів, сигнал спрацьовує тільки вперше. Щоб уникнути несподіванок, важливо перевіряти ще раз умови спрацьовування сигналів при їх використанні в додатку.

За допомогою сигналів створюються ланцюжки. Одна подія – натискання кнопки – може призвести до того, що по черзі відбудуться інші. Ці подальші ефекти відокремлені від того, що їх спричинило. Вони виникають за простими правилами. І це відокремлення ефектів від їхніх тригерів – один із ключових принципів, які враховуються при створенні програм із графічним інтерфейсом.

Ми розглянули сигнали та слоти, показали прості сигнали та їх використання для передачі даних та стану у додатку. Тепер переходимо до віджетів Qt, які будуть використовуватись у додатках разом із сигналами.

Події

Будь-яка взаємодія користувача з програмою Qt – це подія. Є багато типів подій, кожна з яких є окремим типом взаємодії. У події Qt представлені об'єктами подій, в які упакована інформація про подію. Події передаються певним обробникам подій у віджеті, де сталася взаємодія.

Визначаючи користувацькі або розширені обробники подій, можна змінювати спосіб реагування віджетів на них. Обробники подій визначаються так само, як і будь-який інший метод, але назва обробника залежить від типу події, що обробляється.

QMouseEvent – одна з основних подій, що отримуються віджетами. Події QMouseEvent створюються для кожного окремого натискання кнопки миші та її переміщення у віджеті. Ось обробники подій миші:

- **mouseMoveEven**t Миша перемістилася;
- **mousePressEvent** Кнопка миші натиснута;
- **mouseReleaseEvent** Кнопка миші відпущена;
- **mouseDoubleClickEvent** Виявлено подвійний клік.

Наприклад, натискання на віджет призведе до надсилання QMouseEvent в обробник подій **.mousePressEvent** у цьому віджеті.

Події можна перехоплювати, створивши підклас та перевизначивши метод оброблювача у цьому класі. Їх можна фільтрувати, змінювати або ігнорувати, передаючи звичайному обробнику шляхом виклику функції суперкласу методом **super**(). Вони додаються до класу головного вікна наступним чином (у кожному випадку в e буде отримано вхідну подію):

```
import sys
from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import QApplication, QLabel,
QMainWindow, QTextEdit
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.label = QLabel("Click in this window")
        self.setCentralWidget(self.label)

    def mouseMoveEvent(self, e):
        self.label.setText("mouseMoveEvent")
    def mousePressEvent(self, e):
        self.label.setText("mousePressEvent")
```

```
    def mouseReleaseEvent(self, e):
        self.label.setText("mouseReleaseEvent")
    def mouseDoubleClickEvent(self, e):
self.label.setText("mouseDoubleClickEvent")
app = QApplication(sys.argv)
window = MainWindow()
window.show()

app.exec()
```

У вікні перемістіть мишу та натисніть на кнопку, потім двічі натисніть на кнопку та перегляньте, які з'являються події.

Події переміщення миші реєструються лише за натиснутої кнопки. Щоб змінити це, викличте у вікні **self.setMouseTracking(True)**. Події press («Натискання кнопки»), click («Клік») та double-click («Подвійного клік») спрацьовують при натисканні кнопки. Подія release («Відпустити кнопку») спрацьовує, лише коли кнопка відпускається. Клік користувача реєструється зазвичай при натисканні кнопки миші та її відпусканні.

Усередині обробників події є доступ до об'єкта цієї події. Він містить інформацію про подію та використовується, щоб реагувати по-різному залежно від того, що сталося. Розглянемо об'єкти подій керування мишею.

Qt всі події управління мишею відстежуються за допомогою об'єкта **QMouseEvent**. При цьому інформація про подію зчитується з наступних методів подій:

- **.button()** Конкретну кнопку, що викликала цю подію;
- **.buttons()** Стан усіх кнопок миші (прапори OR);
- **.position()** Відносну позицію віджету як цілого QPoint.

Ці методи використовуються в обробнику подій, щоб різні події реагувати по-різному або повністю їх ігнорувати. Через методи позиціонування у вигляді об'єктів QPoint

надається глобальна та локальна (що стосується віджету) інформація про місцезнаходження. Відомості про кнопки надходять із використанням типів кнопок миші з простору імен Qt.

Ієрархія подій

У PyQt кожен віджет - це частина двох різних ієрархій: ієрархії об'єктів у Python та ієрархії макета в Qt. Реакція на події або їх ігнорування впливає на поведінку інтерфейсу користувача.

Часто неохідно перехопити подію, щось із нею зробити, запустивши у своїй поведінка обробки подій за умовчанням. Якщо об'єкт успадкований від стандартного віджету, у нього напевно буде стандартна поведінка. Запустити його можна, методом **super()** викликавши реалізацію із суперкласу. Буде викликаний саме суперклас у Python, а не **.parent()** з PyQt:

```
def mousePressEvent(self, event):
    print("Mouse pressed!")
    super(self,MainWindow).contextMenuEvent(event)
```

Передача вгору за ієрархією макету

Коли в зстосунок додається віджет, він також отримує з макета інший батьківський елемент. Щоб знайти батьківський елемент віджету, необхідно викликати функцію **.parent()**. Іноді ці батьківські елементи вказуються вручну (і часто автоматично), наприклад **QMenu** або **QDialog**. Коли до головного вікна додається віджет, воно стає батьківським елементом віджету.

Коли події створюються для взаємодії користувача з інтерфейсом користувача, вони передаються в його верхній віджет. Якщо у вікні є кнопка і ви натиснете її, вона отримає подію першою.

Якщо перший віджет не може обробити подію, вона перейде до наступного по черзі батьківського віджету. Ця передача вгору по ієрархії вкладених віджетів

продовжиться, поки подія не буде оброблена або досягне головного вікна. В обробниках подій позначають як оброблену через виклик **.accept()**:

```
class CustomButton(QPushButton)
    def mousePressEvent(self, e):
    e.accept()
```

Викликавши в об'єкті події **.ignore()**, позначають її як необроблену. У цьому випадку подія передаватиметься ієрархією вгору:

```
class CustomButton(QPushButton)
    def event(self, e):
    e.ignore()
```

Щоб віджет пропускав події, можна спокійно ігнорувати ті, на які якимось чином реагували. Аналогічно можна відреагувати на одні події, заглушуючи інші.

Віджети

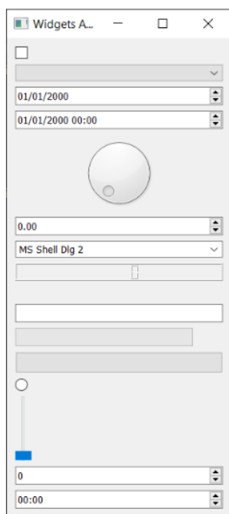
У більшості інтерфейсів користувача і в Qt «віджет» - це компонент, з яким взаємодіє користувач. Інтерфейси користувача складаються з декількох віджетів, розташованих усередині вікна. У Qt є великий вибір віджетів і можна створити власні віджети.

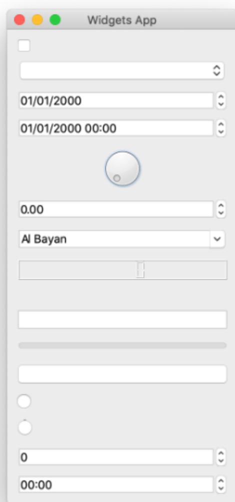
Подивимося на найпоширеніші віджети PyQt. Вони створюються і додаються в макет вікна за допомогою цього коду:

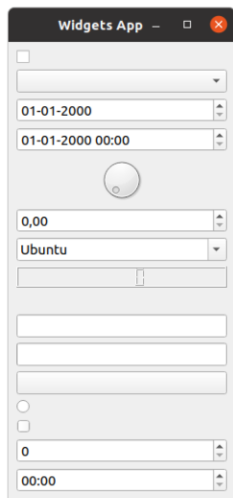
```
import sys
from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import
(QApplication,QCheckBox,QComboBox,QDateEdit,QDateTimeEd
it,QDial,QDoubleSpinBox,QFontComboBox,QLabel,QLCDNumber
,QLineEdit,QMainWindow,QProgressBar,QPushButton,QRadioB
utton,QSlider,QSpinBox,QTimeEdit,QVBoxLayout,QWidget,)

class MainWindow(QMainWindow):
    def __init__(self):
```

```
        super().__init__()
        self.setWindowTitle("Widgets App")
        layout = QVBoxLayout()
        widgets =
[QApplication,QCheckBox,QComboBox,QDateEdit,QDateTimeEd
it,QDial,QDoubleSpinBox,QFontComboBox,QLabel,QLCDNumber
,QLineEdit,QMainWindow,QProgressBar,QPushButton,QRadioB
utton,QSlider,QSpinBox,QTimeEdit]

        for w in widgets:
            layout.addWidget(w())

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)

app = QApplication(sys.argv)
window = MainWindow()
window.show()

app.exec()
```

З'явиться вікно з усіма створеними віджетами:







Короткий опис віджетів:

- QCheckbox Чекбокс;
- QComboBox Вікно списку;
- QDateEdit Для редагування дати та часу;
- QDateTimeEdit Для редагування дати та часу;
- QDial Поворотний циферблат;
- QDoubleSpinbox Спіннер для чисел з плаваючою точкою;
- QFontComboBox Список шрифтів;
- QLCDNumber Досить неприємний дисплей LCD;
- QLabel Просто мітка, не інтерактивна;
- QLineEdit Поле введення з рядком;
- QProgressBar Індикатор виконання;
- QPushButton Кнопка;
- QRadioButton Набір, що перемикається, в якому активний тільки один елемент;
- QSlider Слайдер;
- QSpinBox Спіннер для цілих чисел;
- QTimeEdit Поле редагування часу.

ЧАСТИНА 2. ОСНОВИ WEB-ПРОГРАМУВАННЯ МОВОЮ PYTHON 3

Тема 9. ВСТАНОВЛЕННЯ ТА КОНФІГУРУВАННЯ ФРЕЙМВОРКУ DJANGO. СТВОРЕННЯ DJANGO ПРОЕКТІВ

Django – це високорівневий Python вебфреймворк, що дозволяє швидко створювати безпечні та підтримувані веб-сайти. Він безкоштовний та з відкритим вихідним кодом, має зростаючу та активну спільноту, вичерпну документацію та багато варіантів як платної, так і безкоштовної підтримки.

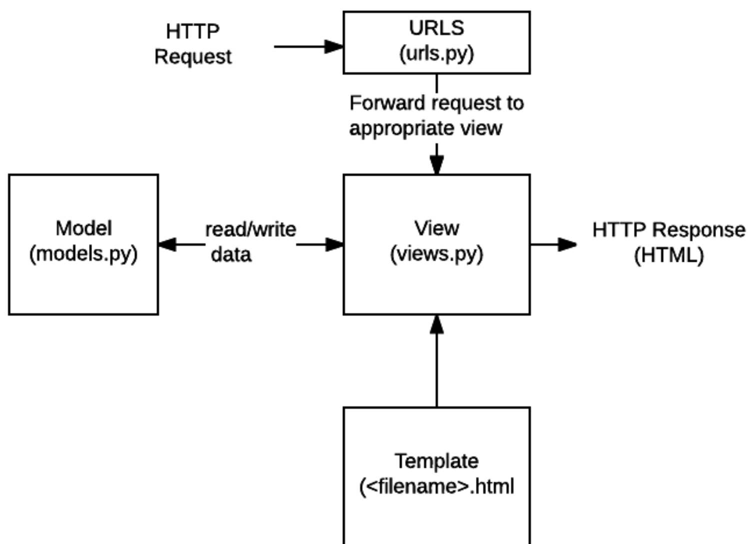
Django був розроблений у період з 2003 по 2005 рік командою, що займалася створенням і обслуговуванням газетних веб-сайтів. Після створення кількох веб-сайтів команда почала повторно використовувати багато спільного коду та шаблонів проектування.

Цей спільний код еволюціонував в веб-фреймворк, який перетворився в проект «*Django*» (на честь музиканта Джанго Рейнарта) з відкритим вихідним кодом у липні 2005 року. *Django* використовують такі великі веб-сайти, як Disqus, Instagram, Knight Foundation, MacArthur Foundation, Mozilla, National Geographic, Open Knowledge Foundation, Pinterest и Open Stack.

Веб-додатки, що написані на *Django*, зазвичай групують код в окремі файли:

- **URLs**: URL-маршрутизатор використовують для перенаправлення HTTP-запитів у відповідне view на основі URL-адреси запита. Крім цього, URL-маршрутизатор може отримувати дані з URL-адреси відповідно до заданого шаблону та передавати їх (дані) у відповідну функцію відображення (view) як аргументи;

- **View**: View (англ. «відображення») – це функція обробника запитів, що отримує HTTP-запити та повертає відповіді. Функція view має доступ до даних, що необхідні для задоволення запитів, та делегує відповіді в шаблони через моделі;
- **Models**: Моделі є об'єктами Python, що визначають структуру даних застосунку та надають механізми для керування (додавання, зміни, видалення) та виконання запитів до бази даних;
- **Templates**: Template (англ. «шаблон») – це текстовий файл, що визначає структуру або розмітку сторінки (наприклад, HTML-сторінки), з полями для підстановки, які використовують для виведення актуального вмісту. View може динамічно створювати HTML-сторінки, використовуючи HTML-шаблони та заповнюючи їх даними з моделі (model). Шаблон може бути використаний для визначення структури файлів будь-яких типів, не обов'язково HTML.



Налаштування середовища розробки Django. Встановлення Python 3.

Ubuntu Linux має Python 3 за замовчуванням.

Аби переконатися у цьому, виконайте у терміналі наступну команду:

```
python3 -V
 Python 3.5.2
```

Проте Python Package Index (pip3) за замовчуванням не встановлено. Для встановлення pip3 у терміналі bash виконайте наступну команду:

```
sudo apt-get install python3-pip
```

Windows не має Python за замовчуванням.

Для встановлення Python виконайте наступні кроки:

1. На веб-сторінці https://www.python.org/downloads/ натисніть кнопку «Download Python 3.x.x».

2. Встановіть Python. Для цього двічі натисніть на файл, що завантажено, та дотримуйтесь інструкції з встановлення.

Окремо встановлювати pip3 (менеджер пакетів Python) не потрібно. pip3 буде встановлено за замовчуванням.

Після цього аби перевірити, що Python успішно встановлено, виконайте наступну команду:

```
py -3 -V
 Python 3.5.2
```

Для перегляду встановлених пакетів Python виконайте наступну команду:

```
pip list
```

Якщо Ви отримали повідомлення про те, що Python не знайдено, то запустіть інсталятор Python знову, оберіть «Modify» та встановіть прапорець «Add Python to environment variables».

Використання Django всередині віртуального середовища Python.

Для створення віртуальних середовищ будемо використовувати бібліотеки *virtualenvwrapper* (Linux та macOS X) та *virtualenvwrapper-win* (Windows). Обидві бібліотеки використовують інструмент *virtualenv*. Окремо встановлювати *virtualenv* не потрібно, оскільки *virtualenvwrapper* та *virtualenvwrapper-win* включають в себе virtualenv.

Для встановлення віртуального середовища в *Ubuntu* виконайте команду:

```
sudo pip3 install virtualenvwrapper
```

Додайте наступні рядки в кінець файлу .bashrc (це прихований файл, що знаходиться у домашній директорії).

```
export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
export VIRTUALENVWRAPPER_VIRTUALENV_ARGS=' -p /usr/bin/python3 '
export PROJECT_HOME=$HOME/Devel
source /usr/local/bin/virtualenvwrapper.sh
```

Далі перезавантажте .bashrc. Для цього виконайте команду:

```
source ~/.bashrc
```

Ви маєте побачити запуск групи скриптів:

```
virtualenvwrapper.user_scripts
creating /home/ubuntu/.virtualenvs/premkproject
virtualenvwrapper.user_scripts
creating /home/ubuntu/.virtualenvs/postmkproject
...
virtualenvwrapper.user_scripts
creating /home/ubuntu/.virtualenvs/preactivate
virtualenvwrapper.user_scripts
creating /home/ubuntu/.virtualenvs/postactivate
virtualenvwrapper.user_scripts
creating /home/ubuntu/.virtualenvs/get_env_details
```

Для встановлення віртуального середовища виконайте наступну команду:

```
pip3 install virtualenvwrapper-win
```

Після встановлення *virtualenvwrapper* чи *virtualenvwrapper-win* робота з віртуальними середовищами дуже схожа на всіх платформах.

Для створення віртуального середовища «*my_django_environment*» виконайте команду:

```
$ mkvirtualenv my_django_environment
```

Зверніть увагу, що всюди далі (тобто далі у цій та у всіх наступних презентаціях з Django) команди виконують у віртуальному середовищі Python.

Деякі корисні команди:

deactivate – вийти з поточного віртуального середовища Python;

workon – переглянути перелік доступних віртуальних середовищ;

workon name_of_environment – активувати вказане віртуальне середовище Python;

rmvirtualenv name_of_environment – видалити вказане віртуальне середовище Python.

Встановлення Django.

Для встановлення Django виконайте команду:

```
pip3 install django
```

Аби перевірити, що Django встановлено, виконайте команду:

```
# Linux/macOS
python3 -m django --version
 3.1.2

# Windows
py -m django --version
 3.1.2
```

Зверніть увагу, що всюди далі для виклику *Python 3* використано команду для *Linux* (*python3*). Якщо Ви працюєте у *Windows*, то замініть цей префікс на: py.

Створення тестового сайту.

У командному рядку чи терміналі перейдіть туди, де бажаєте зберігати Ваші додатки Django. Створіть

директорію для тестового сайту та перейдіть у цю директорію.

```
mkdir django_test
cd django_test
```

Створіть тестовий сайт «mytestsite» за допомогою django-admin:

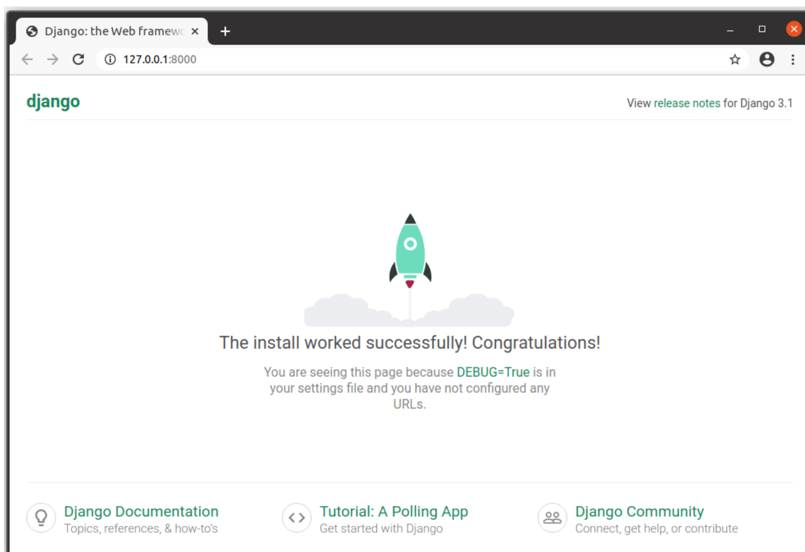
```
django-admin startproject mytestsite
```

Запустіть веб-сервер:

```
cd mytestsite
python3 manage.py runserver
```

Після запуску веб-сервера перейдіть у браузері за посиланням *http://127.0.0.1:8000/*.

Ви маєте побачити сайт, що виглядає наступним чином:



Тема 10. ШАБЛОНИ КОРИСТУВАЦЬКИХ ІНТЕРФЕЙСІВ. ГЕНЕРАЦІЯ ТА ОБРОБКА ФОРМ. ВАЛІДАЦІЯ ДАНИХ

Django - це веб-фреймворк з динамічної генерацією HTML. Найпоширеніший підхід ґрунтується на шаблонах. Шаблон містить статичні частини бажаного представлення HTML, а також деякий спеціальний синтаксис, який описує, як буде вставлено динамічний контент.

Проект Django можна налаштувати з одним або кількома шаблонизаторами (або навіть з жодним, якщо ви не використовуєте шаблони). *Django* надає вбудовані серверні модулі для власної системи шаблонів, що творчо названа мовою шаблонів *Django (DTL)*, і для популярної альтернативи *Jinja2*. Серверні програми для інших мов шаблонів можуть бути доступні для третіх осіб.

Django визначає стандартний API для завантаження та рендерингу шаблонів незалежно від серверної частини. Завантаження складається з пошуку шаблону для даного ідентифікатора та його попередньої обробки, зазвичай компілюючи його в пам'яті. Рендеринг означає інтерполяцію шаблону за допомогою контекстних даних та повернення результуючого рядка.

Припустимо, у нас є проект metanit, і в ньому визначено одну програму - hello:

```
EXPLORER ...
METANIT
  hello
    > __pycache__
    > migrations
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
  metanit
    > __pycache__
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
  db.sqlite3
  manage.py
```

```
settings.py
55  TEMPLATES = [
56      {
57          'BACKEND': 'django.template.backends.django.DjangoTemplates',
58          'DIRS': [],
59          'APP_DIRS': True,
60          'OPTIONS': {
61              'context_processors': [
62                  'django.template.context_processors.debug',
63                  'django.template.context_processors.request',
64                  'django.contrib.auth.context_processors.auth',
65                  'django.contrib.messages.context_processors.messages',
66              ],
67          },
68      },
69  ]
70
71  WSGI_APPLICATION = 'metanit.wsgi.application'
72
73
74  # Database
75  # https://docs.djangoproject.com/en/4.1/ref/settings/#databases
76
```

Налаштування функціональності шаблонів у проекті Django виконується у файлі *settings.py.* за допомогою змінної TEMPLATES. Так, за умовчанням змінна TEMPLATES у файлі *settings.py* має таке визначення:

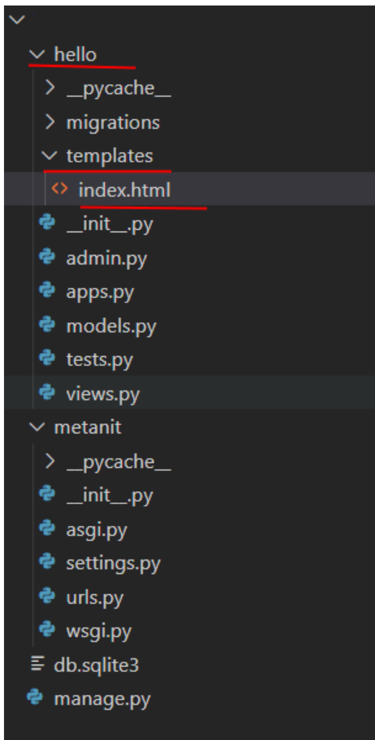
```
TEMPLATES = [
    {
        'BACKEND':
'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',

'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Ця змінна приймає список конфігурацій для кожного рушія шаблонів. За замовчуванням визначено одну конфігурацію, яка має наступні параметри:

- **BACKEND**: рушій шаблонів. двигун шаблонів. За замовчуванням застосовується вбудований рушій *django.template.backends.django.DjangoTemplates*;
- **DIRS**: визначає список каталогів, де рушій шаблонів шукатиме файли шаблонів. За замовчуванням пустий список;
- **APP_DIRS**: вказує, чи рушій шаблонів шукатиме шаблони всередині папок додатків у папці *templates*;
- **OPTIONS**: визначає додатковий перелік параметрів.

Отже, в конфігурації за замовчуванням параметр **APP_DIRS** має значення *True*, а це означає, що рушій шаблонів також шукатиме потрібні файли шаблонів у папці програми в каталозі *templates*. Тобто, за замовчуванням ми вже маємо налаштовану конфігурацію, готову до використання шаблонів. Тепер визначимо самі шаблони.

Додамо до папки програми каталог *templates*, а в ньому визначимо файл *index.html*:



Далі у файлі index.html визначимо наступний код:

```
<!DOCTYPE html>
<html>
<head>
    <title>Django на METANIT.COM</title>
    <meta charset="utf-8" />
</head>
<body>
    <h2>Hello METANIT.COM</h2>
</body>
</html>
```

Це звичайна веб-сторінка, яка містить код *html*. Тепер використовуємо цю сторінку для надсилання відповіді користувачеві. Для цього перейдемо у застосунку *hello* до

файлу *views.py*, який визначає функції обробки запиту. Змінимо цей файл наступним чином:

```
from django.shortcuts import render

def index(request):
    return render(request, "index.html")
```

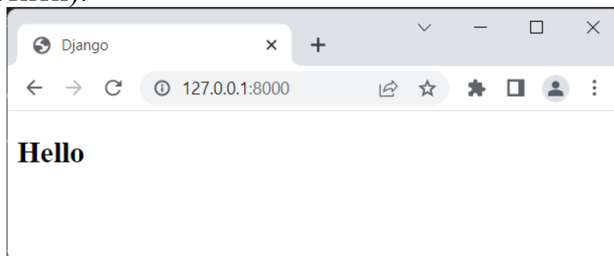
Із модуля *django.shortcuts* імпортується функція **render**.

Функція index викликає функцію *render*, якою передаються об'єкт запиту request та шлях до файлу шаблону в рамках папки templates - "*index.html*".

У файлі **urls.py** проекту пропишемо зіставлення функції *index* із запитом до кореня веб-застосунку:

```
from django.urls import path
from hello import views

urlpatterns = [
    path("", views.index), ]
```

І запустимо проект на виконання і перейдемо до застосунку в браузері (якщо проект запущено, його треба перезапустити):



Файлі **urls.py** має наступний вигляд:

EXPLORER

```
hello
  __pycache__
  migrations
  templates
    index.html
  __init__.py
  admin.py
  apps.py
  models.py
  tests.py
  views.py
metanit
  __pycache__
  __init__.py
  asgi.py
  settings.py
  urls.py
  wsgi.py
db.sqlite3
manage.py
```

views.py

```python
from django.shortcuts import render

def index(request):
    return render(request, "index.html")
```

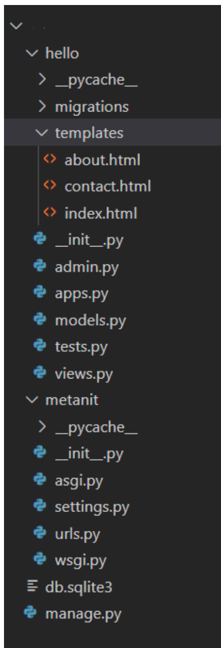
index.html

```html
<!DOCTYPE html>
<html>
<head>
    <title>Django</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>Hello METANIT.COM</h1>
</body>
</html>
```

urls.py

```python
from django.urls import path
from hello import views

urlpatterns = [
    path("", views.index),
]
```

Аналогічно можна вказати й інші шаблони. Наприклад, до папки **templates** додамо ще дві сторінки: *about.html* і *contact.html:*



Також у файлі **views.py** визначимо функції, які використовують дані шаблони:

```
from django.shortcuts import render
 def index(request):
    return render(request, "index.html")
 def about(request):
    return render(request, "about.html")
 def contact(request):
    return render(request, "contact.html")
```

А у файлі **urls.py** зв'яжемо функції з маршрутами:

```
from django.urls import path
from hello import views

urlpatterns = [
    path("", views.index),
    path("about/", views.about),
    path("contact/", views.contact),
]
```

Вище для генерації шаблону застосовувалась функція **render()**, яка є найпоширенішим варіантом. Однак ми також можемо використовувати клас **TemplateResponse**:

```
from django.template.response import
TemplateResponse

def index(request):
    return TemplateResponse(request,  "index.html")
```

Результат буде аналогічним.

Передача даних до шаблонів

Однією з переваг шаблонів і те, що ми можемо передати у яких динамічно з уявлень різні дані. Для виведення даних у шаблоні можуть використовуватись різні способи. Для виведення найпростіших даних застосовується подвійна пара фігурних дужок:

```
{{ object_name }}
```

Наприклад, нехай у проекті ми маємо папку templates, в якій міститься шаблон **index.html**.

Визначимо у файлі **index.html** наступний код:

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
   <title>Django</title>
</head>
<body>
   <h2>{{ header }}</h2>
   <p>{{ message }}</p>
</body>
</html>
```

Тут використовується дві змінні: **message** та **header**. Вони передаватимуться з вистави.

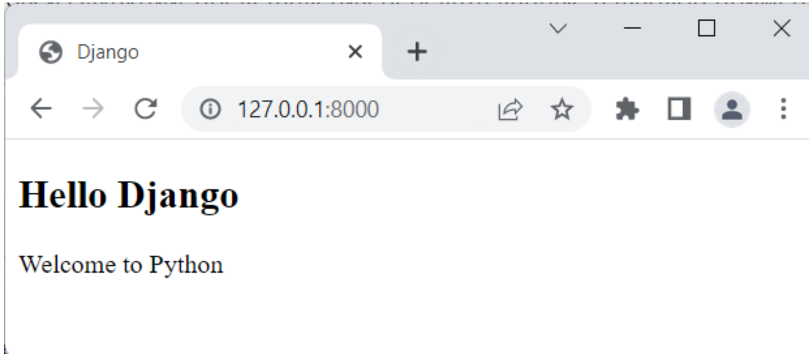
Щоб з функції-представлення передати дані шаблону застосовується третій параметр функції **render**, який ще називається context і який представляє словник. Наприклад, змінимо файл *views.py* наступним чином:

```
from django.shortcuts import render

def index(request):
   data = {"header": "Hello Django", "message":
"Welcome to Python"}
   return render(request, "index.html", context=data)
```

У шаблоні використовуються дві змінні, відповідно словник, який передається у функцію *render* через параметр

context, тепер містить два значення з ключами *header* та *message*.

В результаті при зверненні до кореня веб-додатку ми побачимо наступний висновок у браузері:



Передача складних даних

Розглянемо передачу складніших даних:

```
from django.shortcuts import render

def index(request):
    header="User data"
# звичайна змінна
langs=["Python", "Java", "C#"]            # список
user={"name" : "Tom", "age" : 23}          # словник
address=("Groove street", 23,45)           # кортеж
    data={"header": header, "langs": langs, "user": user,
"address": address}
    return render(request, "index.html", context=data)
```

Як третій параметр у функцію *render* нам треба передати словник, тому всі дані конвертуються в словник і в такому вигляді передаються шаблону.

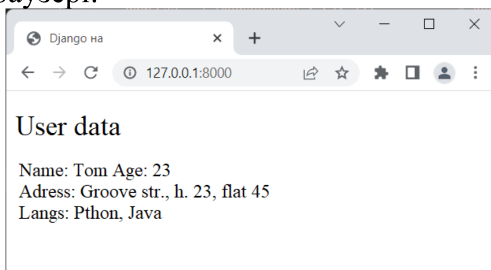
У цьому випадку шаблон міг би виглядати, наприклад, так:

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
   <title>Django</title>
</head>
<body>
   <h1>{{ header }}</h1>
   <p>Name: {{ user.name}} Age: {{user.age}}</p>
   <p>Adress: str. {{address.0}}, h. {{address.1}}, flat.
{{address.2}}</p>
   <p>languages: {{langs.0}}, {{langs.1}}</p>
</body>
</html>
```

Оскільки об'єкти *langs* і *address* представляють відповідно масив і кортеж, то ми можемо звернутися до їх елементів через індекси, як ми б працювали з ними в коді Python, наприклад, перший елемент кортежу *address: address.0.*

Подібним чином, оскільки об'єкт *user* представляє словник, ми можемо звернутися до його елементів за ключами *name* і *age*: {{user.name}} {{user.age}}.

У результаті ми отримаємо наступний вигляд у веббраузері:



Якщо для генерації шаблону застосовується клас **TemplateResponse**, то його конструктор також через третій параметр можна передати дані для шаблону:

```
from django.template.response import
TemplateResponse

def index(request):
    header = "User data"
# звичайна змінна
    langs = ["Python", "Java", "C#"]
# список
    user ={"name" : "Tom", "age" : 23}
# словник
    address = ("Groove str.", 23, 45)
# кортеж

    data = {"header": header, "langs": langs, "user": user,
"address": address}
    return TemplateResponse(request,  "index.html", data)
```

Передача об'єктів класів

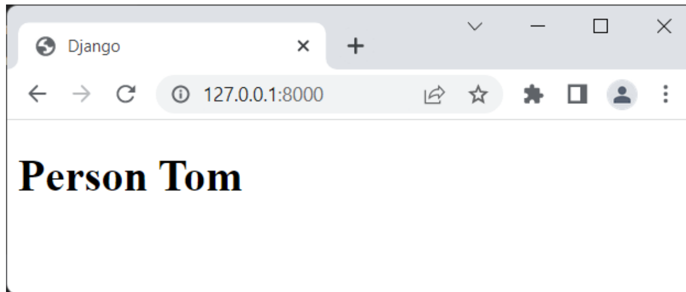
Подібним чином можна передавати шаблони об'єкти своїх класів. Наприклад, визначення функції-представлення:

```
from django.shortcuts import render

  def index(request):
    return render(request, "index.html", context =
{"person": Person("Tom")})

class Person:

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

Тут визначається клас **Person**, конструкторі якого передається деяке значення для атрибута *name*. У функції index шаблон передається об'єкт з ключем "*person*".

У шаблоні index.html ми можемо звертатися до функціональності об'єкта, наприклад до його атрибуту *name*:

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
   <title>Django</title>
</head>
<body>
   <h1>Person {{ person.name }}</h1>
</body>
</html>
```



Форми

HTML-форма — група з одного чи декількох елементів вводу, такі як текстові поля, перемикачі і т.д. та кнопки відправки.

Форми є простим та безпечним методом надсилання даних з клієнту на сервер. Дані надсилаються через запит **POST**, з використанням захисту проти міжсайтової підробки запиту (**CSRF**).

Ім'я:

KNU

Прізвище:

Student

Відправити

Працюючи з формами, необхідно виконати наступні пункти:

- описати форму в HTML;
- перевірити правильність даних з боку клієнта;
- перевірити правильність даних з боку сервера;
- якщо виникла помилка, сповістити користувача;
- в іншому випадку, виконати операції над даними та сповістити користувача.

Більшість цих обов'язків Django бере на себе. Форма та її поля задаються програмним чином, ці об'єкти використовуються як і для генерації HTML-форми, так і для процесу валідації та іншої взаємодії з формою.

Атбрибути HTML-форми

Enter name: Default name for team. OK

```
<form action="/team_name_url/" method="post">
    <label for="team_name">Enter name: </label>
    <input id="team_name" type="text" name="name_field" value="Default name for team.">
    <input type="submit" value="OK">
</form>
```

Атрибути для полів <*input*>:

- **type** — задає тип віджета. Необхідно мати хоча б один елемент типу "submit";

- **id** та **name** — використовуються ідентифікації поля в JS/CSS/HTML;
- **value** — значення поля за замовчуванням.

Атрибут для <*label*>:

- **for** — вказує на id поля, до якого прив'язана мітка.

Атрибути для <*form*>:

- action — URL-адреса, на яку надсилатимуться дані;
- method — HTTP-метод для відправки даних: POST або GET.

POST завжди використовується, якщо відправлення даних призводить до змін в базі даних на сервері. Це підвищує рівень захисту від CSRF.

GET можна використовувати для форм пошукових запитів і т.д.

Процес керування формою з Django наступний:

1. Спочатку клієнт бачить HTML-форму, яка поки що не прив'язана до жодних даних. При тому форма може в цей момент бути або пустою, або містити дані за замовчуванням;
2. Клієнт надсилає форму, і на стороні сервера відбувається зв'язування її даних з класом форми;
3. Це значить, що дані, введені користувачем, а також можливі помилки вводу відносяться саме до цієї форми;
4. Очистка даних (cleanup) — видаляємо невірні символи, які можуть використовуватися для відправки шкідливих даних на сервер (sanitization), а також перетворення текстових даних в типи даних Python;
5. Валідація (validation) — перевірка значень полів, наприклад, на правильність введених дат чи їх діапазонів;

6. Якщо очистка та валідація пройшли успішно, то виконуємо необхідні дії з даними (наприклад, зберігаємо їх в базу даних), та перенаправляємо користувача на іншу сторінку;
7. Якщо дані не пройшли перевірку, то знову зображуємо форму, вставляємо введені користувачем дані, але на цей раз додаємо повідомлення про помилки.

Покажемо процес створення сторінки, яка дозволить, наприклад, бібліотекарю оновляти інформацію про книги (зокрема, ввести дату повернення книги).

Для цього створимо форму з елементом вводу дати, задамо ініціальне значення (дата через 3 тижні з сьогоднішнього дня).

Валідація: введена дата не повинна бути в минулому, або в далекому майбутньому.

Після всіх перевірок запишемо значення у поле *BookInstance.due_back*.

Клас Form

Клас Form визначає:

- поля форми;
- показ віджетів та текстових міток;
- початкові значення;
- валідація значень;
- сповіщення щодо помилок.

Також цей клас надає методи для відображення себе в шаблоні, у передбачених форматах (таблиці, списки і т.д.), або для отримання значень будь-якого елемента (що дозволяє виконувати більш точне відображення).

```
from django import forms

class RenewBookForm(forms.Form):
    renewal_date = forms.DateField(help_text="Enter a date between now and 4 weeks (default 3).")
```

Тут бачимо найпростішу імплементацію класу Form.

Структура класу Form схожа на клас Model, і використовує аналогічні типи полів (наприклад, *BooleanField*, *CharField*, *EmailField*, *ImageField*, і.д.)

В загальному випадку об'єкти **Field* приймають такі аргументи: *required* (чи обов'язкове для вводу), *label* (текстова мітка), initial (початкове значення), *help_text* (додатковий текст), *widget* (віджет, що застосується для поля), тощо.

Валідація

```
def clean_renewal_date(self):
    data = self.cleaned_data['renewal_date']

    if data < datetime.date.today():
        raise ValidationError(_('Invalid date - renewal in past'))

    if data > datetime.date.today() + datetime.timedelta(weeks=4):
        raise ValidationError(_('Invalid date - renewal more than 4 weeks ahead'))

    return data
```

Валідація відбувається завдяки методам *clean_<назва_поля>*() (де *назва_поля* — це назва поля, яке перевіряємо).

В нашому випадку переконуємось, що введена дата не є у минулому, і не є в далекому майбутньому (більш, ніж 4 тижні).

Метод повертає "очищені" дані.

Текст перекладено на мову користувача завдяки функції перекладу Django *ugettext_lazy()*, її імпортуємо як _():

```
from django.core.exceptions import ValidationError
from django.utils.translation import ugettext_lazy as _
import datetime
```

Відображення форми

```
def renew_book_librarian(request, pk):
    book_inst = get_object_or_404(BookInstance, pk=pk)

    if request.method == 'POST':
        form = RenewBookForm(request.POST)

        if form.is_valid():
            book_inst.due_back = form.cleaned_data['renewal_date']
            book_inst.save()

            return HttpResponseRedirect(reverse('all-borrowed') )
    else:
        proposed_renewal_date = datetime.date.today() + datetime.timedelta(weeks=3)
        form = RenewBookForm(initial={'renewal_date': proposed_renewal_date,})

    return render(request, 'catalog/book_renew_librarian.html', {'form': form, 'bookinst':book_inst})
```

Додамо наступну функцію у файл **views.py**:

pk (*primary key*) — ідентифікатор книги. Функція *get_object_or_404* знаходить цей об'єкт в моделі, або ж видає помилку "Не знайдено".

Для методу **POST**:

Виконуємо валідацію, якщо все добре, то зберігаємо дані у модель, і направляємо користувача на наступну сторінку (*HttpResponseRedirect*).

URL-адресу отримуємо через URL конфігурацію завдяки функції **reverse**.

Якщо дані не пройшли валідацію, то дані залишаються в формі, яка потім рендериться (*render*). На цей раз Django разом з формою покаже користувачу повідомлення про помилку.

Для методу **GET**:

Просто покажемо користувачу форму. Додамо початкові дані, вони задаються параметром конструктора *initial*.

Функція-декоратор *@permission_required* обмежує доступ до відображення. Лише користувачі з дозволом '*can_mark_returned*' зможуть скористатися цією формою.

Конфігурація URL-адреси

У файлі **locallibrary/catalog/urls.py** матимемо таке:

```
urlpatterns += [
    url(r'^book/(?P<pk>[-\w]+)/renew/$', views.renew_book_librarian, name='renew-book-librarian'),
]
```

Задано регулярний вираз (*RegEx*), який розпізнає всі адреси у форматі */catalog/book/<ідентифікатор BookInstance>/renew/* і направить їх до функції *renew_book_librarian* у файлі *views.py*.

Ідентифікатор *BookInstance* буде знаходитись у групі під назвою pk (*primary key*), саме він і надасться як параметр функції. Повинен виглядати як UUID (наприклад, *123e4567-e89b-12d3-a456-426655440000*).

Створюємо шаблон, який посилається на наше відображення **/catalog/templates/catalog/book_renew_librarian.html:**

```
{% extends "base_generic.html" %}
{% block content %}

    <h1>Renew: {{bookinst.book.title}}</h1>
    <p>Borrower: {{bookinst.borrower}}</p>
    <p{% if bookinst.is_overdue %} class="text-danger"{% endif %}>Due date: {{bookinst.due_back}}</p>

    <form action="" method="post">
        {% csrf_token %}
        <table>
        {{ form }}
        </table>
        <input type="submit" value="Submit" />
    </form>

{% endblock %}
```

Варто звернути увагу на елемент <form>.

Задано метод **POST**, оскільки ця форма модифікує базу даних. Необхідно забезпечити безпеку вебдодатку.

Обов’язково додати блок *{% csrf_token %}*, він є частиною фреймворку Django та служить для боротьби з атакою **CSRF**.

Змінна {{ *form* }} містить власне форму.

```
<form action="" method="post">
    {% csrf_token %}
    <table>
    {{ form }}
    </table>
    <input type="submit" value="Submit" />
</form>
```

Змінна {{ *form* }} містить кожне поле форми у окремому рядку таблиці *(<tr>)*.

```
<tr>
  <th><label for="id_renewal_date">Renewal date:</label></th>
  <td>
    <input id="id_renewal_date" name="renewal_date" type="text" value="2016-11-08" required />
    <br />
    <span class="helptext">Enter date between now and 4 weeks (default 3 weeks).</span>
  </td>
</tr>
```

Якщо під час валідації форми виявилися помилки, то отримаємо також повідомлення про помилку:

```
<tr>
  <th><label for="id_renewal_date">Renewal date:</label></th>
   <td>
      <ul class="errorlist">
        <li>Invalid date - renewal in past</li>
      </ul>
      <input id="id_renewal_date" name="renewal_date" type="text" value="2015-11-08" required />
      <br />
      <span class="helptext">Enter date between now and 4 weeks (default 3 weeks).</span>
    </td>
</tr>
```

Аналогічного результату (рядки таблиці) можна отримати, використавши змінну {{ form.as_table }}.

Також можна зобразити кожне поле як список елементів ({{ *form.as_ul* }}), або як параграф ({{ *form.as_p* }}).

Більш точно контролювати рендеринг форми можна завдяки дот-нотації (точки). Наприклад:

- {{*form.renewal_date*}}: власне поле;
- {{*form.renewal_date.errors*}}: список помилок, якщо вони є;

- {{*form.renewal_date.help_text*}}: допоміжний текст, і т.д.

Ця форма матиме наступний вигляд:

127.0.0.1:8000/catalog/book/d2ad0f63-82b0-46ac-8511-c89b76756a6b/renew/

Home
All books
All authors

User: superman
My Borrowed
Logout

Staff
All borrowed

Renew: The Wise Man's Fear

Borrower: superman

Due date: Oct. 12, 2016

Renewal date: 2016-11-08
Enter date between now and 4 weeks (default 3 weeks).

Submit

Початкове значення дати користувач бачить, вперше зайшовши на сторінку.

У разі виникнення помилки:

127.0.0.1:8000/catalog/book/d2ad0f63-82b0-46ac-8511-c89b76756a6b/renew/

Home
All books
All authors

User: superman
My Borrowed
Logout

Staff
All borrowed

Renew: The Wise Man's Fear

Borrower: superman

Due date: Oct. 12, 2016

- Invalid date - renewal in past

Renewal date: 2015-11-08
Enter date between now and 4 weeks (default 3 weeks).

Submit

Клас ModelForm

Клас Form є досить гнучким: можна задати форму будь-якого вигляду та використовувати її для складних операцій. Наприклад, форму можна пов’язати з кількома моделями одночасно.

В нашому випадку наша форма відповідає одній і тільки одній моделі *BookInstance*. Насправді, ця модель вже містить всі необхідні дані для створення простих форм (поля, їх типи, мітки і т.д.)

В таких випадках, аби уникнути дублцікації коду, можна скористатися спрощеним класом *ModelForm*.

Цього коду достатньо, щоб створити форму, асоційовану з моделлю *BookInstance*:

```
from django.forms import ModelForm

from catalog.models import BookInstance

class RenewBookModelForm(ModelForm):
    class Meta:
        model = BookInstance
        fields = ['due_back']
```

У підкласі *Meta* зазначаємо клас моделі та список необхідних полів.

Можливо, властивості полів, надані моделлю (їх назви, мітки, тощо) не зовсім підходять для конкретної задачі.

Наприклад, потрібно підібрати кращу назву для поля ("*New renewal date*" замість "*Due back*"). Тоді в клас *Meta* додаємо:

```
class Meta:
    model = BookInstance
    fields = ['due_back']
    labels = {'due_back': _('New renewal date')}
    help_texts = {'due_back': _('Enter a date between now and 4 weeks (default 3).')}
```

Метод *clean_**, тепер цей клас є повністю ідентичним до попередньої імплементації *Form*:

```
class RenewBookModelForm(ModelForm):
    def clean_due_back(self):
       data = self.cleaned_data['due_back']

       # Check if a date is not in the past.
       if data < datetime.date.today():
           raise ValidationError(_('Invalid date - renewal in past'))

       # Check if a date is in the allowed range (+4 weeks from today).
       if data > datetime.date.today() + datetime.timedelta(weeks=4):
           raise ValidationError(_('Invalid date - renewal more than 4 weeks ahead'))

       # Remember to always return the cleaned data.
       return data

    class Meta:
        model = BookInstance
        fields = ['due_back']
        labels = {'due_back': _('Renewal date')}
        help_texts = {'due_back': _('Enter a date between now and 4 weeks (default 3).')}
```

Узагальнені класи відображень

Якщо ж потрібно додати до сайту інструменти адмністрування, а саме надати бібліотекарям можливість додавати, створювати та видаляти авторів, Це можна зробити за допомогою трьох класів Form та трьох класів View.

Django об'єднує в один клас функціонал і Form, і View.

Узагальнені класи відображень - абстрактні класи, що визначають відображення для деяких дуже поширених "*use case*", а саме:

- створення об'єктів;
- редагування об'єктів;
- видалення об'єктів.

Досить легко додати в *views.py* імплементації трьох узагальнених класів.

Аналогічно до *ModelForm* задаємо потрібні поля та інші атрибути, наприклад, початкові значення.

Варто звернути увагу на параметр *success_url* - він вказує на те, куди перенаправити користувача після відправлення форми.

Перегляди *Create* та *Update* автоматично направлять користувача на сторінку з деталями на відповідного об'єкта *Author*.

В цьому випадку для задання *success_url* використовується не *reverse*, а *reverse_lazy*. Необхідно використовувати "ліниве" обчислення, бо узагальнені класи завантажуються до того, як фіналізується URL-конфігурація.

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.urls import reverse_lazy

from catalog.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['first_name', 'last_name', 'date_of_birth', 'date_of_death']
    initial = {'date_of_death': '11/06/2020'}

class AuthorUpdate(UpdateView):
    model = Author
    fields = '__all__'

class AuthorDelete(DeleteView):
    model = Author
    success_url = reverse_lazy('authors')
```

Для узагальнених класів відображень необхідно задати шаблони.

Відображення "*create*" та "*update*" за замовчуванням використовують один і той самий шаблон, назва якого походить від назви моделі: **<назва_моделі>_form.html**

```
{% extends "base_generic.html" %}

{% block content %}
  <form action="" method="post">
    {% csrf_token %}
    <table>
    {{ form.as_table }}
    </table>
    <input type="submit" value="Submit">
  </form>
{% endblock %}
```

Створимо такий шаблон *author_form.html*:

Так само, як і раніше, показуємо дані у формі таблиці, та не забуваємо про *{% csrf_token %}*.

Відображення “*delete*” потребує шаблон для підтвердження операції видалення, під назвою **<назва_моделі>_confirm_delete.html**:

```
{% extends "base_generic.html" %}

{% block content %}

<h1>Delete Author</h1>

<p>Are you sure you want to delete the author: {{ author }}?</p>

<form action="" method="POST">
  {% csrf_token %}
  <input type="submit" value="Yes, delete.">
</form>

{% endblock %}
```

Додамо перегляди у URL-конфігурацію:

```
urlpatterns += [
    path('author/create/', views.AuthorCreate.as_view(), name='author-create'),
    path('author/<int:pk>/update/', views.AuthorUpdate.as_view(), name='author-update'),
    path('author/<int:pk>/delete/', views.AuthorDelete.as_view(), name='author-delete'),
]
```

AuthorCreate, AuthorUpdate та AuthorDelete є класами, тому отримуємо перегляд через метод *as_view()*. Узагальнені класи для редагування та видалення обов'язково потребують параметр під назвою **pk (primary key).**

Форми для створення та редагування авторів виглядатимуть наступним чином:

127.0.0.1:8000/catalog/author/create/

Home
All books
All authors

User: superman
My Borrowed

First name:
Last name:
Date of birth:
Died: 12/10/2016
Submit

:

В свою чергу, форма для підтвердження видалення виглядатиме так:

127.0.0.1:8000/catalog/author/10/delete/

Home
All books
All authors

User: superman
My Borrowed
Logout

Delete Author

Are you sure you want to delete the author: Twain, Mark?

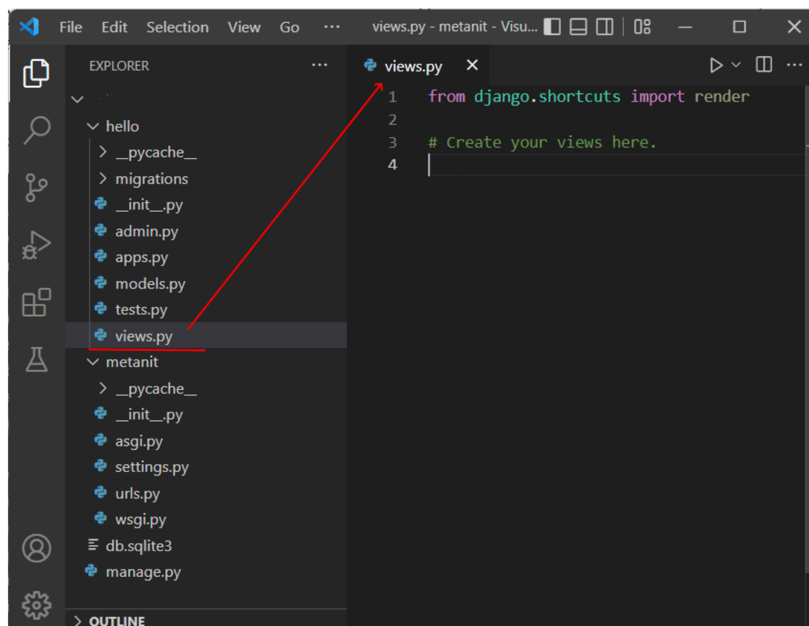
Yes, delete.

Тема 11. КОНТРОЛЕРИ ТА ОБРОБКА ЗАПИТІВ КОРИСТУВАЧІВ. РОБОТА З ДИНАМІЧНИМИ ДАНИМИ. КУКИ. СЕСІЇ. ФІЛЬТРИ

Незмінним елементом будь-якого вебдодатку є обробка запитів, які відправляє користувач. У Django за обробки запитів відповідають представлення або перегляди. По суті представлення представляють функції обробки, які приймають дані запиту у вигляді об’єкта **HttpRequest** з пакета **django.http** і генерують інший результат, який потім надсилається користувачеві.

За замовчуванням представлені розміщені в розміщенні у файлі **views.py**.

Наприклад, розглянемо стандартний проект, у який додано застосунок:



При створенні нового проекту файл **views.py** має наступне вміст:

from django.shortcuts import render

Create your views here.

Даний код поки що не обробляє запити, він лише імпортує функцію **render()**, яка може використовуватись для обробки.

Генерувати результат можна різними способами. Один із них представляє використання класу **HttpResponse** з пакета **django.http**, який дозволяє відправити текстове вміст.

Так, змінимо файл **views.py** наступним чином:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Головна")

def about(request):
    return HttpResponse("Про сайт")

def contact(request):
    return HttpResponse("Контакти")
```

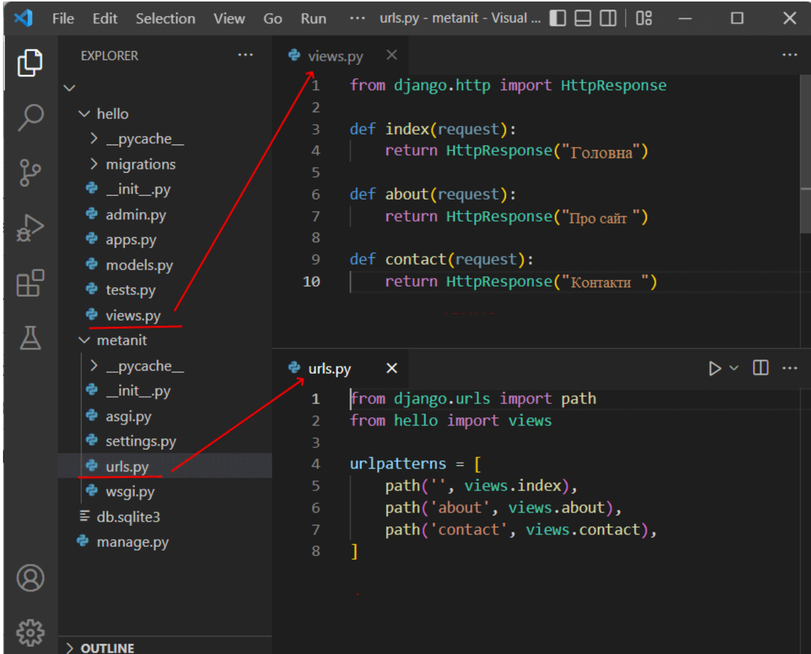
У даному коді визначені три функції, які будуть обробляти запити. Будь-яка функція приймає в якості параметра запиту об'єкт **HttpRequest**, який зберігає інформацію про запит. Однак у цьому випадку вона нам не потрібна, тому параметр ніяк не використовується. Для генерації відповіді в конструктор об'єкта **HttpResponse** передається деяка строка. Це може бути в тому числі і код **html** у вигляді рядка.

Щоб ці функції складалися із запитами, необхідно визначити для них маршрути в проекті у файлі **urls.py**. Зокрема, змінюємо цей файл таким чином:

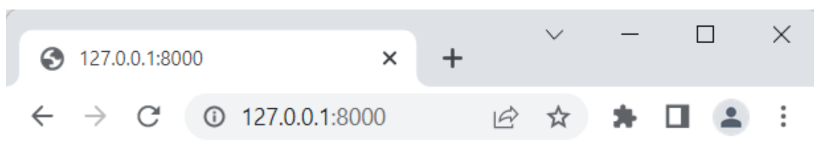
```
from django.urls import path
from hello import views

urlpatterns = [
    path('', views.index),
    path('about', views.about),
```

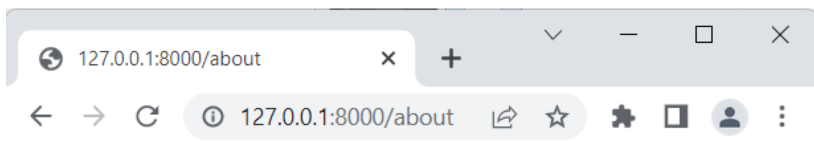
```
        path('contact', views.contact),
    ]
```

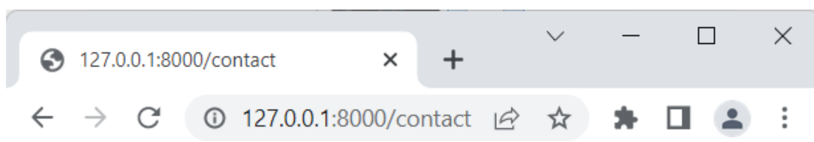
Змінна **urlpatterns** визначає набір поставлених функцій обробки з певними рядками запиту. Наприклад, запит в корені веб-сайту буде оброблятися функцією **index**, запит за адресою "*about*" буде оброблятися функцією *about*, а запит "*contact*" - функцією *contact*.



Запустимо проект і звернемося за деякими із цих адрес:





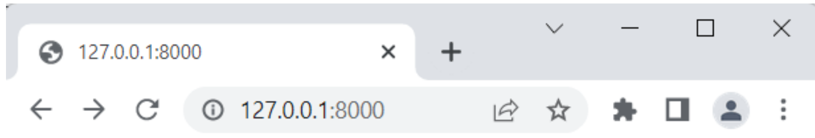


При цьому ми можемо надіслати не простий текст, а, наприклад, код *html*, який потім інтерпретується браузером. Так, змінимо файл **views.py** наступним чином:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("<h2>Головна</h2>")
```

```
def about(request):
    return HttpResponse("<h2>Про сайт</h2>")

def contact(request):
    return HttpResponse("<h2>Контакти</h2>")
```

Відповідно тепер браузер отримає код *html*, наприклад:



Отримання даних запиту

Функції-представлення в якості обов'язкового параметра отримують об'єкт **HttpRequest**, який зберігає інформацію про запит. **HttpRequest** визначає ряд атрибутів, які зберігають інформацію про запит. Найважливішими є наступні:

- **scheme**: схема запиту (http або https);
- **body**: представляє тело запиту у вигляді строки байтів;
- **path**: представляє шлях запиту;
- **method**: метод запиту (GET, POST, PUT і т.д.);
- **encoding**: кодування;
- **content_type**: тип вмісту запиту (значення заголовка CONTENT_TYPE);
- **GET**: об'єкт у вигляді словника, який містить параметри запиту GET;

- **POST**: об'єкт у вигляді словника, який містить параметри запиту POST;
- **COOKIES**: відправлені клієнтом куки;
- **FILES**: відправлені клієнтом файли;
- **META**: зберігає всі доступні заголовки **http** у вигляді словника;
- **headers**: заголовки запиту у вигляді словника.

Також HttpRequest визначає низку методів. Зазначимо такі:

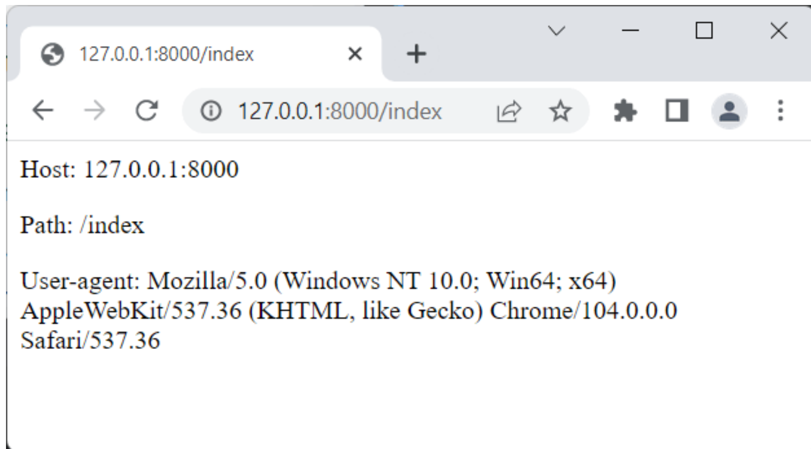
- **get_full_path()**: повертає повний шлях запиту, включаючи рядок запиту;
- **get_host()**: повертає хост клієнта, для цього використовується значення заголовків HTTP_X_FORWARDED_HOST (якщо включена опція USE_X_FORWARDED_HOST) та HTTP_HOST;
- **get_port()**: повертає номер порту.

Наприклад, отримаємо деяку інформацію про запит. Для цього у файлі **views.py** внесемо наступні зміни:

```
from django.http import HttpResponse

def index(request):
    host = request.META["HTTP_HOST"]
# отримуємо адреси серверів
    user_agent =
request.META["HTTP_USER_AGENT"]
# отримуємо дані браузера
    path = request.path
# отримуємо адресу

    return HttpResponse(f"""
      <p>Host: {host}</p>
      <p>Path: {path}</p>
      <p>User-agent: {user_agent}</p>
```

""")

Результат виконання:



В даному випадку одержуємо два заголовки "HTTP_HOST" та "HTTP_USER_AGENT" та відповідний шлях.

У файлі **urls.py** запишемо цю функцію:

```
from django.urls import path
from hello import views

urlpatterns = [
    path("index", views.index),
]
```

HttpResponse та надсилання відповіді

Для надсилання відповіді клієнту Django застосовується клас **HttpResponse** з пакету **django.http**. У загальному випадку для відправлення деяких даних достатньо передати дані в конструктор **HttpResponse**. Наприклад, нехай у файлі **views.py** є найпростіша функція-подання, яка надсилає відповідь клієнту:

```
from django.http import HttpResponse
```

```
def index(request):
    return HttpResponse("Hello")
```

Та у файлі **urls.py** ця функція співвідноситься з деяким маршрутом:

```
from django.urls import path
from hello import views

urlpatterns = [
    path("", views.index),
]
```

Подібна функція просто передає **HttpResponse** деякий текст, який користувач потім побачить у браузері. Однак подібною функціональністю **HttpResponse** не обмежується. Так, функція ініціалізації класу визначає кілька параметрів:

```
HttpResponse.__init__(content=b'', content_type=None,
status=200, reason=None, charset=None, headers=None)
```

Параметри функції наступні:

- **content**: вміст відповіді у вигляді рядка байтів. Якщо передається інший вміст, він конвертується в рядок байтів;
- **content_type**: MIME-тип відповіді, встановлює HTTP-заголовок Content-Type. Якщо цей параметр не встановлений, то застосовується mime-тип text/html і значення DEFAULT_CHARSET, тобто в результаті буде: "text/html; charset=utf-8";
- **charset**: кодування відповіді у вигляді рядка. За промовчанням django намагається встановити кодування з параметра content_type, а в разі невдачі для встановлення кодування застосовується налаштування DEFAULT_CHARSET;
- **status**: статусний код відповіді. За замовчуванням дорівнює 200;

- **reason_phrase**: повідомлення, яке надсилається разом статусним кодом;
- **headers**: заголовки відповіді у вигляді словника.

Для зберігання даних визначено ряд атрибутів. Деякі з них:

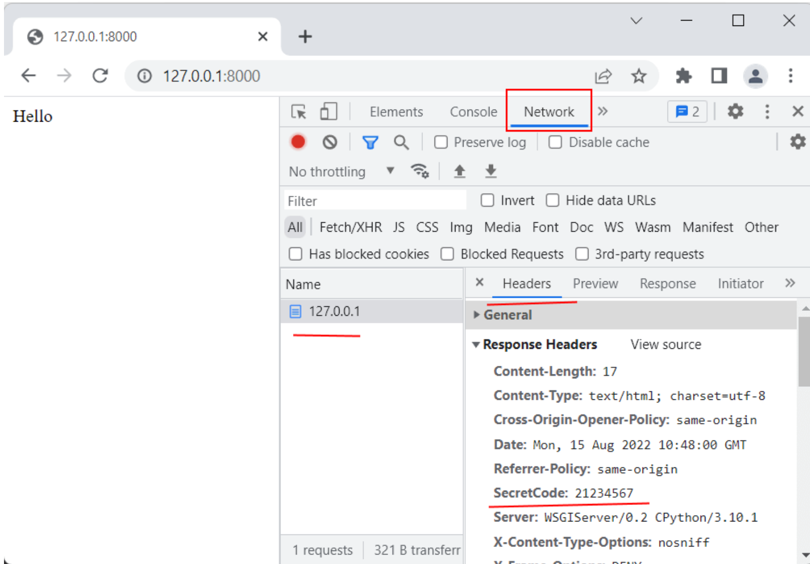
- **content**: вміст відповіді у вигляді рядка байтів;
- **headers**: надіслані заголовки у вигляді словника;
- **charset**: кодування відповіді у вигляді рядка. За замовчуванням django намагається встановити кодування із заголовка content_type, а в разі невдачі для встановлення кодування застосовується налаштування DEFAULT_CHARSET;
- **status_code**: статусний код відповіді;
- **reason_phrase**: повідомлення, яке надсилається разом статусним кодом.

Розглянемо деякі можливості. Наприклад, змінимо визначення функції у файлі views.py:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello", headers={"SecretCode": "21234567"})
```

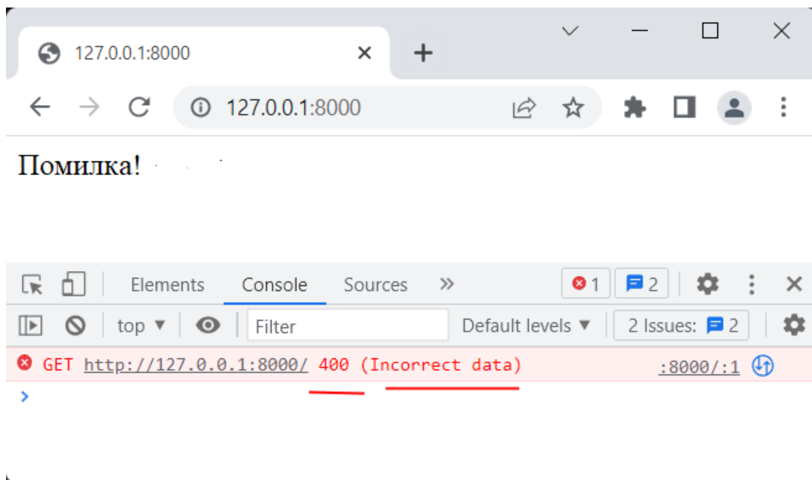
У цьому випадку також встановлюється заголовок "*SecretCode*". Хоча насправді в HTTP немає такого заголовка, але ми можемо визначати кастомні заголовки, щоб передати через них клієнту якусь інформацію. Наприклад, після звернення до функції ми можемо в браузері через інструменти розробника проінспектувати

надіслані сервером заголовки і знайти там, зокрема, заголовок "*SecretCode*".



Подібним чином можна задати інші параметри, наприклад, код статусу та повідомлення для нього:

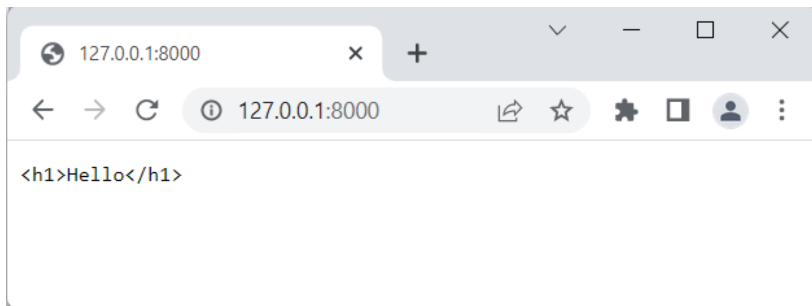
```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Помилка!", status=400,
reason="Incorrect data")
```



Встановлення вмісту та кодування:

```
from django.http import HttpResponse
 def index(request):
    return HttpResponse("<h1>Hello</h1>", 
content_type="text/plain", charset="utf-8")
```

Хоча у вмісті відповіді застосовуються теги html (<h1>), але браузер тепер буде розглядати цей вміст як простий текст, тому що встановлений заголовок "**text/plain**":



Переадресація та надсилання статусних кодів

При переміщенні документа з однієї адреси на іншу ми можемо скористатися механізмом переадресації, щоб

вказати користувачам та пошуковику, що документ тепер доступний за новою адресою.

Переадресація буває тимчасова та постійна. При тимчасовій переадресації ми вказуємо, що документ тимчасово переміщено на нову адресу. У цьому випадку у відповідь надсилається статусний код 302. При постійній переадресації ми повідомляємо, що документ тепер буде доставлено за новою адресою

Для створення тимчасової переадресації застосовується клас **HttpResponseRedirect**, а для постійної – клас **HttpResponsePermanentRedirect**, які розташовані у пакеті **django.http**.

Так, визначимо у файлі views.py наступний код:

```
from django.http import HttpResponse,
HttpResponseRedirect, HttpResponsePermanentRedirect


def index(request):
    return HttpResponse("Index")

def about(request):
    return HttpResponse("About")

def contact(request):
    return HttpResponseRedirect("/about")

def details(request):
    return HttpResponsePermanentRedirect("/")
```

При зверненні до функції contact вона буде перенаправляти шляхом "*about*", який оброблятиметься функцією *about*. А функція details використовуватиме постійну переадресацію і перенаправлятиме на корінь вебзастосунки.

Також у файлі **urls.py** для тестування визначимо наступні маршрути:

```
from django.urls import path
from hello import views
```

```
urlpatterns = [
    path("", views.index),
    path("about/", views.about),
    path("contact/", views.contact),
    path("details/", views.details),
]
```

Надсилання статусних кодів

У пакеті django.http є ряд класів, які дозволяють надсилати певний код статуса:

Статусний код	Клас
304 (Not Modified)	HttpResponseNotModified
400 (Bad Request)	HttpResponseBadRequest
403 (Forbidden)	HttpResponseForbidden
404 (Not Found)	HttpResponseNotFound
405 (Method Not Allowed)	HttpResponseNotAllowed
410 (Gone)	HttpResponseGone
500 (Internal Server Error)	HttpResponseServerError

У функцію конструктора цих класів можна передати деякі дані, наприклад повідомлення про помилку, яке побачить користувач:

```
HttpResponseNotModified()
HttpResponseBadRequest("Bad Request")
HttpResponseForbidden("Forbidden")
HttpResponseNotFound("Not Found")
HttpResponseNotAllowed("Method is not allowed")
HttpResponseGone("Content is no longer here")
HttpResponseServerError("Server Error")
```

Наприклад, створимо наступний файл **views.py**:

```
from django.http import HttpResponse,
HttpResponseNotFound, HttpResponseForbidden,
HttpResponseBadRequest

def index(request, id):
    people = ["Tom", "Bob", "Sam"]
    # якщо користувача знайдено, повертаємо його
    if id in range(0, len(people)):
        return HttpResponse(people[id])
    # якщо ні, повертаємо помилку 404
    else:
        return HttpResponseNotFound("Not Found")

def access(request, age):
    # якщо вік не входитьв діапазон 1-400, повертаємо
помилку 400
    if age not in range(1, 111):
        return HttpResponseBadRequest("Некоректні
дані")
    # якщо вік більше 17, то доступ дозволено
    if(age > 17):
        return HttpResponse("Доступ разрешен")
    # якщо ні, повертаємо помилку 403
```

```
        else:
            return HttpResponseForbidden("Доступ заблоковано: недостатній вік")
```

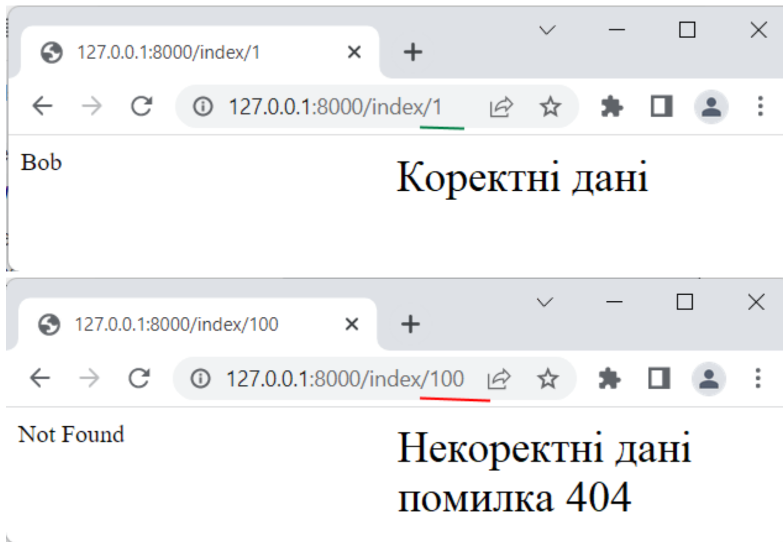
Функція index приймає id. Це буде індекс елемента у списку людей. Однак користувач може передати і недійсний індекс, наприклад, 100, хоча у прикладі у списку лише 3 елементи. Тому, якщо передано дійсний індекс, то повертаємо елемент цього індексу. Якщо ж індекс недійсний, то за допомогою класу **HttpResponseNotFound** повертаємо помилку 404 та повідомлення про помилку.

Аналогічно функція access приймає параметр age, який представляє умовний вік користувача. Якщо вік виходить за деякі розумні межі (1-110), то за допомогою класу HttpResponseBadRequest повертаємо помилку 400. Якщо вік вкладається в ці рамки, але він менший за 18, то за допомогою класу HttpResponseForbidden повертаємо помилку 403 про те, що доступ заборонено.

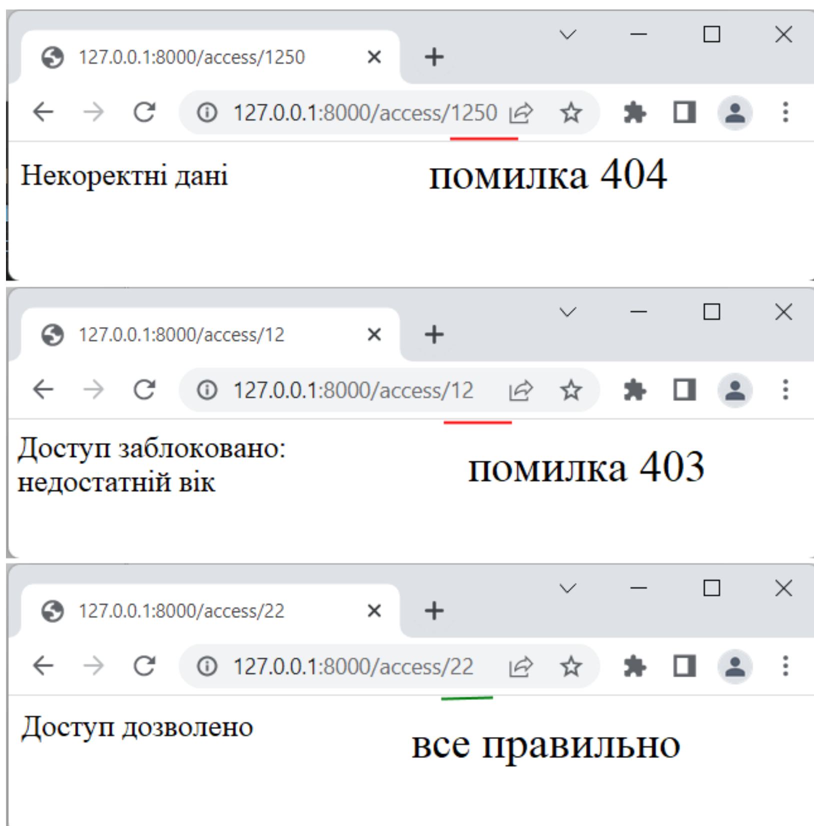
Для тесту у файлі **urls.py** визначимо такі маршрути:

```
from django.urls import path
from hello import views
urlpatterns = [
    path("index/<int:id>", views.index),
    path("access/<int:age>", views.access), ]
```

Звернувшись до функції *index*, передавши для параметра id коректні, та некоректні значення, в залежності від значення параметра ми побачимо або дані зі списку людей, або помилку 404:



Аналогічно при зверненні на адресу "/*access*" в залежності від значення параметра *age* ми побачимо або повідомлення про помилку, або звичайне повідомлення:



Варто зазначити, що статусні повідомлення про помилки також відображаються в консоль запущеної програми.

Надсилання та отримання кук

Куки (*Cookie*) є найпростішим способом зберегти дані користувача. Куки зберігаються на комп'ютері користувача

і можуть встановлюватися як на сервері, так і клієнта. Так як куки посилаються з кожним запитом на сервер, їх максимальний розмір обмежений 4096 байтами. Розглянемо, як відправити клієнту і отримати від клієнта кукі в застосунку Django.

Встановлення куки

За встановлення cookie та відправлення їх клієнту відповідає метод **set_cookie()** класу **HttpResponse**. Цей метод має наступний вигляд:

set_cookie(key, value='', max_age=None, expires=None, path='/', domain=None, secure=False, httponly=False, samesite=None)

Параметри методу:

- **key**: ключ або ім'я кукі;
- **value**: значення куки;
- **max_age**: максимальний час життя кукі в секундах. Це може бути об'єкт timedelta, чи число, чи значення None (обмежує час життя куки поточної сесією браузера, є значенням за промовчанням);
- **expires**: час і дата, коли закінчується дія кукі. Повинен представляти рядок у форматі "Wdy, DD-Mon-YY HH:MM:SS GMT" або об'єкт datetime.datetime;
- **path**: шлях, для якого встановлюються куки;
- **domain**: домен, до якого застосовуються куки;
- **secure**: встановлює протокол, що використовується. Так, якщо має значення True, то куки надсилатимуться на сервер лише у запиті за протоколом https;
- **httponly**: встановлює доступність для скриптів JavaScript на клієнті. Так, значення httponly=True запобігає доступу до cookie з коду javascript на клієнті;

- **samesite**: встановлює дозволи на відправлення куки в кросдоменні запити. Так, значення samesite='Strict' і samesite='Lax' вказують браузеру не посилати куки в кросдоменні запити. Значення за промовчанням samesite='None' дозволяє надсилати куки в кросдоменні запити.

Також клас HttpResponse надає ще один метод для встановлення кукі:

```
set_signed_cookie(key, value, salt='', max_age=None,
expires=None, path='/', domain=None, secure=False,
httponly=False, samesite=None)
```

Цей метод приймає самі параметри, його відмінність у цьому, що він застосовує шифрування. Необов'язковий параметр **salt** дозволяє встановити сіль для шифрування.

Наприклад, встановимо кукі. Нехай у файлі **views.py** є така функція:

```
from django.http import HttpResponse

# встановлюємо кукі
def set(request):
    # отримуємо з рядка запиту користувача
    username = request.GET.get("username",
"Undefined")
    # створюємо відповідь
    response = HttpResponse(f"Hello {username}")
    # передаємо його в кукі
    response.set_cookie("username", username)
    return response
```

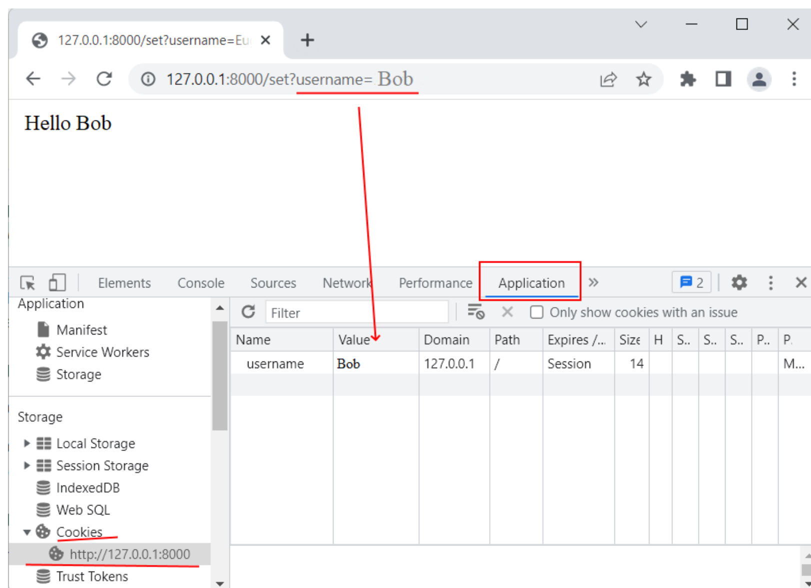
З рядка запиту отримуємо значення параметра "username" і передаємо його в кукі за однойменним ключем

Нехай у файлі **urls.py** ця функція викликається у запиті за шляхом "/set":

```
from django.urls import path
from hello import views
```

```
urlpatterns = [
    path("set", views.set),
]
```

Звернемося до адреси "/set", передавши через рядок запиту параметр username, і сервер встановить куки, а браузер збереже їх. Зможемо їх побачити через інструменти розробника:



Отримання кукі

Якщо кукі не шифровані, для їх отримання в об'єкті **HttpRequest** можна використовувати атрибут **COOKIES**, який представляє словник.

Якщо куки шифровані методом **set_signed_cookie**, то їх отримання в класі HttpRequest застосовується метод:

get_signed_cookie(key, default=RAISE_ERROR, salt='', max_age=None)

Параметри методу:

key: ключ куки, які треба отримати;

default: значення за промовчанням, якщо куки із зазначеним ключем відсутні у запиті;

max_age: максимальний час життя кукі в секундах;

salt: тип шифрування, повинен мати те саме значення, яке передавалося в set_signed_cookie.

Наприклад, отримаємо раніше встановлені куки по ключу. Визначимо у файлі **views.py** додаткову функцію **get()**:

```
from django.http import HttpResponse

# встановлюємо кукі
def set(request):
    # отримуємо з рядка запиту користувача
    username = request.GET.get("username",
"Undefined")
    response = HttpResponse(f"Hello {username}")
    # передаємо його в кукі

    response.set_cookie("username", username)
    return response

# отримуємо кукі
def get(request):
    # отримуєм кукі з ключем username

    username = request.COOKIES["username"]
    return HttpResponse(f"Hello {username}")
```

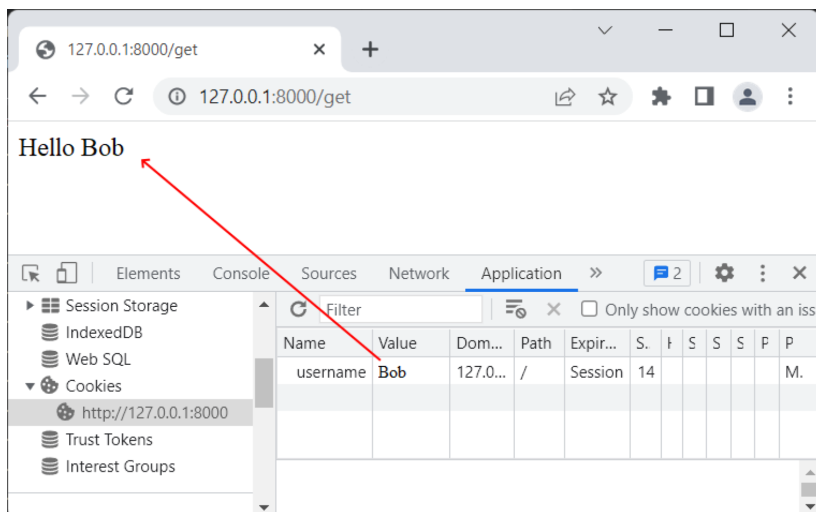
За допомогою виразу request.COOKIES["username"] отримуємо куки за ключом "username" і передаємо їх значення у відповідь клієнту.

Нехай у файлі **urls.py** ця функція викликається у запиті шляхом "/get":

```
from django.urls import path
from hello import views

urlpatterns = [
```

```
        path("set", views.set),
        path("get", views.get),
    ]
```

Звернемося на адресу "/get" і отримаємо раніше встановлені кукі *username*:



Сесії

Сесії дозволяють реалізувати такий вид функціональності, який надасть змогу зберігати й отримувати довільні дані, одержані на основі індивідуальної поведінки користувача на сайті.

Вся взаємодія між браузерами і серверами здійснюється за допомогою протоколу HTTP, який не зберігає свій стан (**stateless**).

Сесія є механізмом для відслідковування "**стану**" між сайтом і якимось браузером. Сесії дозволяють зберігати будь-які дані браузера і отримувати їх в той момент, коли між браузером і сайтом встановиться з'єднання. Дані отримуються і зберігаються в сесії за допомогою відповідного "**ключа**".

Django використовує кукі, які містять спеціальний ідентифікатор сесії. За замовчуванням дані сесії зберігаються в базі даних сайту (це безпечніше, ніж зберігати в кукі, де вони вразливі для зловмисників). Проте можливо налаштувати Django так, щоб зберігати дані сесії в інших місцях (кеші, файлах, "безпечних" кукі).

Сесії доступні з моменту створення бази сайту. Необхідні конфігурації виконуються в розділах *INSTALLED_APPS* і *MIDDLEWARE* файлу проекту (**locallibrary/locallibrary/settings.py**), як показано нижче:

```
INSTALLED_APPS = [
    ...
    'django.contrib.sessions',

MIDDLEWARE = [
    ...
'django.contrib.sessions.middleware.SessionMiddleware',
    ....
```

Отримати доступ до змінної **session** у відповідному відображенні можливо через параметр *request* (**HttpRequest** передається як перший аргумент в кожне відображення). Змінна сесії є зв’язком з конкретним користувачем (і, якщо бути точнішим, зв’язком з певним браузером, який визначається за допомогою ідентифікатора (id) сесії, отримуваного з кукі браузера).

Змінна *session* є об’єктом-словником, який служить для читання й запису необмеженої кількості разів. З нею можливо виконувати будь-які стандартні операції, включаючи очистку всіх даних, перевірку наявності ключа, цикли по даним і тд.

Нижче фрагмент коду показує як задавати і видаляти деякі дані за допомогою ключа "*my_car*", зв’язаного з поточною сесією:

```
# Отримання значення сесії за допомогою
ключа(тобто, 'my_car').
```

```
# Якщо такого немає, то виникне помилка KeyError
my_car = request.session['my_car']

# Отримання значення сесії. Якщо значення не існує,
# то повернеться значення за замовчуванням ('mini')
my_car = request.session.get('my_car', 'mini')

# Передача значення в сесію
request.session['my_car'] = 'mini'

# Видалення значення з сесії
del request.session['my_car']
```

Особливістю Django є те, що не потрібно задумуватись про механізм, який зв'язує сесію з поточним запитом у відображенні: my_car зв'язана з браузером, який відправив запит.

Дане **API** має й інші методи, які переважно використовуються для керування кукі, пов'язаних із сесією. Наприклад, існують методи перевірки того, що кукі підтримуються клієнтським браузером, інші методи слугують для установки і перевірки граничних дат життя кукі, а також для очищення прострочених сесій зі сховища.

За замовчуванням Django зберігає дані сесії в базу даних і відправляє відповідні кукі клієнту тільки тоді, коли сесія була змінена, або видалена.

```
# Дане присвоєння розпізнається як оновлення сесії
# і дані будуть збережені
request.session['my_car'] = 'mini'
```

Якщо оновлювати інформацію всередині даних сесії, то Django не розпізнає ці зміни і не збереже їх. В таких випадках потрібно явно вказати, що сесія була змінена.

```
# Об'єкт сесії модифікується неявно. Зміни НЕ
БУДУТЬ збережені!
request.session['my_car']['wheels'] = 'mini'
```

```
# Сесія буде збережена, кукі оновлені (якщо необхідно).
 request.session.modified = True
```

Також можливо змінити поведінку сесії таким чином, щоб сайт оновлював базу даних і відправляв кукі при кожному запиті, додавши **SESSION_SAVE_EVERY_REQUEST = True** у файл налаштування проекту (locallibrary/locallibrary/settings.py).

Для прикладу, на базі простго сайту додамо функціонал повідомлення користувачу кількості здійснених ним візитів головної сторінки сайту LocalLibrary. Для цього в **views.py** необхідно додати такі зміни:

```
def index(request):
    ...

num_authors=Author.objects.count()
# all()' визначено за замовчуванням

# Кількість візитів
num_visits=request.session.get('num_visits', 0)
request.session['num_visits'] = num_visits + 1

# Рендеринг HTML прототипу index.html з даними змінної.
    return render(
        request,
        'index.html',
context={'num_books':num_books,'num_instances':num_instances,
'num_instances_available':num_instances_available,'num_authors':num_authors,
'num_visits':num_visits}, # num_visits appended
        )
```

В першу чергу буде отримано значення *'num_visits'* із сесії, повертаючи 0, якщо воно не було встановлено раніше.

Кожен раз при отриманні запиту, збільшуватиметься дане значення на одиницю і зберігатиметься в сесію (до наступного візиту даної сторінки користувачем). Далі змінна num_visits передається в шаблон через змінну context. Для показу значення змінної потрібно додати такий код в шаблон (*index.html*), в його нижній розділ "**Dynamic content**":

```
	<h2>Dynamic content</h2>
	<p>The library has the following record counts:</p>
	<ul>
	<li><strong>Books:</strong> {{ num_books }}</li>
	<li><strong>Copies:</strong> {{ num_instances
}}</li>
	<li><strong>Copies available:</strong> {{
num_instances_available }}</li>
	<li><strong>Authors:</strong> {{ num_authors
}}</li>
	</ul>
	<p>You have visited this page {{ num_visits }}
	{% if num_visits == 1 %} time{% else %} times{%
endif %}.</p>
```

Тема 12. МОДЕЛІ ТА АКТИВНІ ЗАПИСИ. ПРЕДСТАВЛЕННЯ, ЗБЕРЕЖЕННЯ ТА ОБРОБКА ДАНИХ. МІГРАЦІЇ. АСОЦІАЦІЇ ТА ВІДНОШЕННЯ МІЖ ТИПАМИ

Вебзастосунки Django отримують доступ і керують даними через об'єкти Python, які називаються моделями. Моделі визначають структуру даних, що зберігаються, включаючи типи полів і, можливо, їх максимальний розмір, значення за замовчуванням, параметри списку вибору, текст довідки для документації, текст міток для форм і т. д. Визначення моделі не залежить від основної бази даних - можливо вибрати один з кількох компонентів вашого налаштування проекту. Після того, як ви обрали якусь базу даних хочете використовувати, вам не потрібно безпосередньо працювати з нею - ви просто пишете свою структуру моделі та код, а Django робить всю роботу, пов'язану з базою даних.

Розглянемо побудову моделі на прикладі сайту бібліотеки LocalLibrary.

Необхідно зберігати інформацію про книжках (назва, резюме, автор, мову, якою написана книга, категорія, ISBN) і що ми можебути кілька доступних примірників (з унікальним глобальним ідентифікатором, статусом доступності тощо. буд.). Нам може знадобитися зберігати більше інформації про автора, ніж просто їх ім'я, і може бути кілька авторів з однаковими чи схожими іменами. Важливо мати можливість сортувати інформацію на основі назви книги, автора, писемної мови та категорії.

При проектуванні ваших моделей має сенс мати окремі моделі для кожного об'єкта (група зв'язаної інформації). У цьому випадку очевидними об'єктами є книги, екземпляри книг та автори.

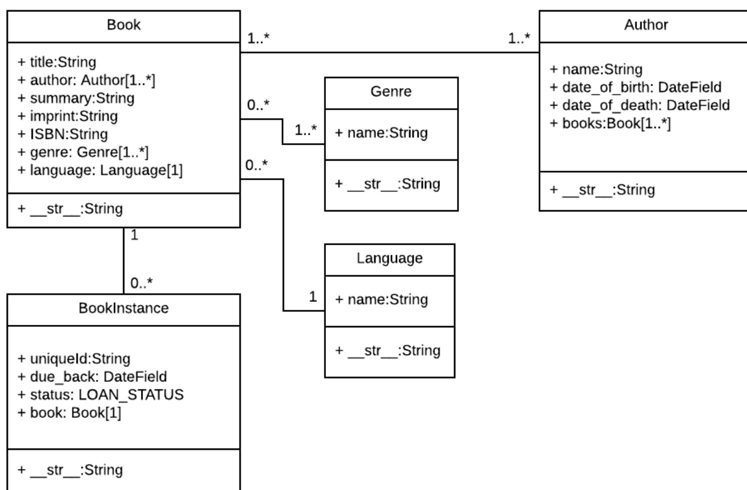
Ви також можете використовувати моделі для представлення параметрів списку вибору (наприклад, як випадний список варіантів) замість жорсткого кодування

вибору на самому вебсайті – це рекомендується, коли всі варіанти невідомі заздалегідь або можуть змінитися. Очевидні кандидати на моделі в цьому випадку включають жанр книги (наприклад, "Наукова фантастика", "Французька поезія" і т. д.) та мова (англійська, французька, японська).

Як тільки ми визначилися з нашими моделями та полями, нам треба подумати про стосунки. Django дозволяє вам визначати відносини як один до одного (**OneToOneField**), один до багатьох (**ForeignKey**) і багато до багатьох (**ManyToManyField**).

Діаграма асоціації UML, наведена нижче, показує моделі, які ми визначили в цьому випадку (у вигляді блоків). Як і вище, ми створили моделі для книги (загальні відомості про книгу), екземпляр книги (статус конкретних фізичних копій книги, доступних у системі) та автора. Ми також вирішили створити модель для жанру, щоб можна було створювати/вибирати значення через інтерфейс адміністратора . Ми вирішили не мати модель для BookInstance: status – ми жорстко закодували значення (**LOAN_STATUS**), тому що ми не очікуємо їх зміни. У кожному з полів можна побачити ім'я моделі, імена та типи полів, а також методи та їх типи повернення.

На діаграмі також показані залежності між моделями, включаючи їх множники. Множники є числами на діаграмі, що показують мінімум і максимум одиниць кожної моделі, які можуть бути у цьому зв'язку. Наприклад, сполучна лінія між ящиками показує, що книга та жанр пов'язані між собою. Цифри, близькі до моделі жанру, показують, що у книги може бути один або кілька жанрів (скільки завгодно), а числа на іншому кінці рядка поряд з моделлю книги показують, що жанр може мати нуль або більше пов'язаних книг.



Нижче наведено короткий огляд того, як визначається модель, і деякі з найважливіших полів і аргументів поля.

Визначення моделі

Моделі зазвичай визначаються у додатку **models.py**. Вони реалізуються як підкласи **django.db.models.Model**, і можуть включати поля, методи та метадані. У наведеному нижче фрагменті коду показано «типову» модель, названу **MyModelName**:

```
from django.db import models

class MyModelName(models.Model):
    # Поля
    my_field_name = models.CharField(max_length=20, help_text="Enter field documentation")
    ...

    # Метадані
    class Meta:
        ordering = ["-my_field_name"]
```

```
    # Методи
    def get_absolute_url(self):
        """
        Повертає URL для конкретного MyModelName.
        """
        return reverse('model-detail-view',
args=[str(self.id)])

    def __str__(self):
        """
        Рядок для представлення MyModelName (на
сайты Адміна)
        """
        return self.field_name
```

Далі наступних розділах ми докладно розглянемо кожен елемент усередині моделі

Поля

Модель може мати довільну кількість полів будь-якого типу – кожен представляє стовпець даних, який ми хочемо зберегти в одній із наших таблиць бази даних. Кожен запис (рядок) бази даних буде складатися з одного значення кожного поля. Розглянемо наведений вище приклад:

```
my_field_name = models.CharField(max_length=20,
help_text="Enter field documentation")
```

Наведений вище приклад має одне поле **my_field_name**, типу **models.CharField** - що означає, що це поле міститиме рядки буквено-цифрових символів. Типи полів призначаються з використанням певних класів, які визначають тип запису, який використовується для зберігання даних у базі даних, а також критерії перевірки,

які повинні використовуватися, коли значення отримані з форми HTML (тобто становить дійсне значення). Типи полів можуть приймати аргументи, які додатково визначають, як поле зберігається або може використовуватися. У цьому випадку ми даємо нашому полю два аргументи:

- **max_length=20** – вказує, що максимальна довжина значення цього поля становить 20 символів;
- **help_text="Enter field documentation"** – надає текстову мітку для відображення, щоб допомогти користувачам дізнатися, яке значення необхідно надати, коли це значення має бути введено користувачем через HTML-форму.

Ім'я поля використовується для звернення до нього у запитах та шаблонах. У полях також є мітка, яка задається як аргумент (**verbose_name**), або виводиться шляхом великої літери першої літери імені змінної поля та заміни будь-яких символів підкреслення пропуском (наприклад, **my_field_name** матиме мітку за умовчанням **My field name**).

Порядок, в якому оголошуються поля, впливатиме на їхній порядок за замовчуванням, якщо модель відображається у формі (наприклад, на сайті адміністратора), хоча це може бути перевизначено.

Наступні загальні аргументи можуть використовуватися при оголошенні багатьох типів полів:

- **help_text**: Надає текстову мітку для HTML-форм (наприклад, на сайті адміністратора), як описано вище;
- **verbose_name**: Ім'я, що використовується для поля, що використовується в полі мітки. Якщо не вказано, Django виведе за промовчанням докладну назву від імені поля;

- **default**: Стандартне значення для поля. Це може бути значення або об'єкт, що викликається, і в цьому випадку об'єкт буде викликатися щоразу, коли створюється новий запис;
- **null**: Якщо True, Django буде зберігати порожні значення як NULL в базі даних для полів, де це доречно (CharField натомість збереже порожній рядок). За промовчанням використовується значення False;
- **blank**: Якщо true, поле може бути порожнім у ваших формах. За промовчанням використовується значення False, що означає, що перевірка форми Django змусить вас ввести значення. Це часто використовується з null = True, тому що якщо ви хочете дозволити порожні значення, ви також хочете, щоб база даних могла представляти їх належним чином;
- **choices**: Група варіантів цього поля. Якщо це передбачено, відповідний віджет форми за промовчанням буде полем вибору з цими варіантами замість стандартного текстового поля;
- **primary_key**: Якщо True, задає поточне поле як первинний ключ для моделі (первинний ключ - це спеціальний стовпець бази даних, призначений для однозначної ідентифікації всіх різних записів таблиці). Якщо в якості первинного ключа не вказано поле, Django автоматично додасть для цього поле.

Наступні загальні аргументи можуть використовуватися при оголошенні багатьох / різних полів:

- **CharField** використовується для визначення рядків фіксованої довжини від короткої до

середньої. Ви повинні вказати max_length для зберігання даних;

- **TextField** використовується для великих рядків довільної довжини. Ви можете вказати max_length для поля, але це використовується лише тоді, коли поле відображається у формах (воно не застосовується на рівні бази даних).
- **IntegerField** це поле для зберігання значень (цілого числа) і для перевірки введених значень у вигляді цілих чисел у формах;
- **DateField** та **DateTimeField** використовуються для зберігання/подання дат та інформації про дату/час (як Python datetime.date та datetime.datetime, відповідно). Ці поля можуть додатково оголошувати (взаємовиключні) параметри auto_now=True (для встановлення поля на поточну дату кожного разу, коли модель зберігається), auto_now_add (тільки для встановлення дати, коли модель була вперше створена) та за замовчуванням (щоб встановити дату за умовчанням, яку користувач може перевстановити);
- **EmailField** використовується для зберігання та перевірки адрес електронної пошти;
- **FileField** та ImageField використовуються для завантаження файлів та зображень відповідно (ImageField просто додає додаткову перевірку, що завантажений файл є зображенням). Вони мають параметри визначення того, як і де зберігаються завантажені файли;
- **AutoField** особливий тип IntegerField, який автоматично збільшується. Первинний ключ цього типу автоматично додається до вашої моделі, якщо ви явно не вкажете його;
- **ForeignKey** використовується для вказівки відношення "один до багатьох" до іншої моделі

бази даних (наприклад, автомобіль має одного виробника, але виробник може робити багато автомобілів). "Одна" сторона відносини - це модель, що містить ключ.

Існує багато інших типів полів, включаючи поля для різних типів чисел (великі цілі числа, малі цілі числа, дробові), логічні значення, URL-адреси, slugs, унікальні ідентифікатори та інші «пов'язані з часом» відомості (тривалість, час тощо).

Метадані

Можливо оголосити метадані на рівні моделі для своєї моделі, оголосивши клас **Meta**:

```
class Meta:
   ordering = ["-my_field_name"]
   ...
```

Однією з найбільш корисних функцій цих метаданих є керування сортуванням записів, що повертаються при запиті типу моделі. Ви можете зробити це, вказавши відповідність назви полів для сортування, як показано вище. Порядок буде залежати від типу поля (поля символів відсортовані за абеткою, а поля дати відсортовані в хронологічному порядку). Як показано вище, ви можете префікс імені поля мінус-символом (-), щоб змінити порядок сортування.

Наприклад, щоб сортувати книги за замовчуванням:

```
ordering = ["title", "-pubdate"]
```

Книги будуть відсортовані за алфавітом за назвою, від A-Z, а потім за датою публікації всередині кожної назви, від найновішої до найстарішої.

Інші корисні атрибути дозволяють створювати та застосовувати нові «дозволи доступу» для моделі (дозволи за замовчуванням застосовуються автоматично), дозволити впорядкування на основі іншого поля або оголосити, що клас є «абстрактним» (базовий клас, для якого ви не можете

створювати записи, та натомість буде створено для створення інших моделей). Багато інших параметрів метаданих керують тим, яка база даних повинна використовуватися для моделі та як зберігаються дані (це дійсно корисно, якщо вам потрібно зіставити модель з наявною базою даних).

Методи

Модель може мати методи. Мінімально в кожній моделі ви повинні визначити стандартний метод класу для **Python __str __ ()**, щоб повернути рядок для кожного об'єкта. Цей рядок використовується для представлення окремих записів на сайті адміністрування. Часто це повертає поле назви чи імені з моделі:

```
def __str__(self):
    return self.field_name
```

Іншим поширеним методом включення до моделі Django є **get_absolute_url()**, яка повертає URL-адресу для відображення окремих записів моделі на веб-сайті (якщо ви визначаєте цей метод, тоді Django автоматично додасть кнопку «Перегляд на сайті» на екранах редагування записів моделі на сайті адміністратора). Типовий шаблон для **get_absolute_url()** показаний нижче:

```
def get_absolute_url(self):
    """
    Повертає URL для доступу до конкретного стану
моделі
    """
    return reverse('model-detail-view', args=[str(self.id)])
```

Після того, як ви визначили свої класи моделей, ви можете використовувати їх для створення, оновлення або видалення записів і запуску запитів для отримання всіх записів або окремих підмножин записів.

Створення та зміна записів

Щоб створити запис, можна визначити екземпляр моделі, а потім викликати метод **save ()**:

```
a_record = MyModelName(my_field_name="Instance #1")
a_record.save()
```

Ви можете отримати доступ до полів у цьому новому записі за допомогою синтаксису точок та змінити значення. Ви повинні викликати **save ()**, щоб зберегти зміни в базі даних:

```
print(a_record.id)
print(a_record.my_field_name)

a_record.my_field_name="New Instance Name"
a_record.save()
```

Визначення моделей LocalLibrary

Почнемо визначати моделі бібліотеки. Відкрийте **models.py** (в /locallibrary/catalog/). Шаблон у верхній частині сторінки імпортує модуль моделей, що містить базовий клас моделі **models.Model**, від якого успадковуються наші моделі.

```
from django.db import models
```

Модель жанру:

Ця модель використовується для зберігання інформації про категорію книг - наприклад, чи то художня чи документальна, роман чи військово-історична тощо. буд. значення могли управлятися через базу даних, а чи не були закодованими. Вставте наведений нижче код моделі Genre в нижню частину файлу **models.py**.

```
class Genre(models.Model):

    name = models.CharField(max_length=200, help_text="Enter a book genre (e.g. Science Fiction, French Poetry etc.)")
```

```
    def __str__(self):

        return self.name
```

Модель має один **CharField field** (ім'я), яке використовується для опису жанру (воно обмежене 200 символами і має деякий **help_text**. Наприкінці моделі ми оголошуємо метод **__str__()**, який просто повертає ім'я жанру, визначеного конкретним записом. **Verbose name** не було визначено, тому поле називатиметься **Name** у формах.

Модель книги:

Скопіюйте модель книги нижче та знову вставте її в нижню частину файлу. Модель книги представляє всю інформацію про доступну книгу в загальному розумінні, але не конкретний фізичний «примірник» або «копію» для тимчасового використання. Модель використовує **CharField** для представлення назви книги та isbn (зверніть увагу, як isbn вказує свій ярлик як «ISBN», використовуючи перший неназваний параметр, оскільки в іншому випадку ярлик за умовчанням був би «Isbn»). Модель використовує **TextField** для **summary**, тому що цей текст, можливо, має бути дуже довгим:

```
from django.urls import reverse #Used to generate URLs
by reversing the URL patterns

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey('Author',
on_delete=models.SET_NULL, null=True)

    summary = models.TextField(max_length=1000,
help_text="Enter a brief description of the book")
    isbn = models.CharField('ISBN',max_length=13,
help_text='13 Character <a href="https://www.isbn-
international.org/content/what-isbn">ISBN number</a>')
```

```
        genre = models.ManyToManyField(Genre, help_text="Select a genre for this book")

        def __str__(self):
            return self.title


        def get_absolute_url(self):
            return reverse('book-detail', args=[str(self.id)])
```

Жанр представляє собою **ManyToManyField**, так що книга може мати кілька жанрів, а жанр може мати багато книг. Автор оголошується через **ForeignKey**, тому в кожній книзі буде тільки один автор, але автор може мати багато книг.

В обох типах полів відповідний клас моделі оголошується як перший неіменований параметр, використовуючи клас моделі, або рядок, що містить ім'я відповідної моделі. Ви повинні використовувати ім'я моделі як рядок, якщо пов'язаний клас ще не був визначений у цьому файлі, перш ніж він буде вказаний! Іншими параметрами, які становлять інтерес для поля автора, є null=True, що дозволяє базі даних зберігати значення **Null**, якщо автор не вибраний, і **on_delete = models**. **SET_NULL** встановить значення автора в **Null**, якщо пов'язаний з автором запис буде видалено.

Модель також визначає __str __ (), використовуючи поле заголовка книги для представлення книги. Остаточний метод **get_absolute_url ()** повертає URL-адресу, яку можна використовувати для доступу до докладного запису для цієї моделі (для цього нам потрібно буде визначити зіставлення URL-адрес, у якому міститься докладна інформація про книгу, і визначити пов'язане уявлення та шаблон).

Модель BookInstance:

Скопіюйте модель **BookInstance** (наведено нижче) під інші моделі. **BookInstance** являє собою певну копію

книги, яку хтось може позичати, і включає інформацію про те, чи доступна копія або в який день вона очікується, «відбиток» або відомості про версію, а також унікальний ідентифікатор книги в бібліотеці. Тепер деякі з полів та методів будуть знайомі. Модель використовує:

- **ForeignKey** для ідентифікації пов'язаної книги (у кожній книзі може бути багато копій, але в копії може бути лише одна книга);
- **CharField**, для представлення даних (конкретного випуску) про книгу.

```
import uuid # Required for unique book instances

class BookInstance(models.Model):

    id = models.UUIDField(primary_key=True,
default=uuid.uuid4, help_text="Unique ID for this particular
book across whole library")
    book = models.ForeignKey('Book',
on_delete=models.SET_NULL, null=True)
    imprint = models.CharField(max_length=200)
    due_back = models.DateField(null=True, blank=True)

    LOAN_STATUS = (
        ('m', 'Maintenance'),
        ('o', 'On loan'),
        ('a', 'Available'),
        ('r', 'Reserved'),
    )

    status = models.CharField(max_length=1,
choices=LOAN_STATUS, blank=True, default='m',
help_text='Book availability')

    class Meta:
        ordering = ["due_back"]
```

```
def __str__(self):
    return '%s (%s)' % (self.id,self.book.title)
```

Додатково оголошуємо кілька нових типів полів:

- **UUIDField** використовується для поля ID, щоб встановити його як primary_key для цієї моделі. Цей тип поля виділяє глобальне унікальне значення для кожного екземпляра (по одному для кожної книги, яку ви можете знайти у бібліотеці);
- **DateField** використовується для даних due_back (при яких очікується, що книга з'явиться після запозичення чи обслуговування). Це значення може бути blank або null (необхідно, коли книга доступна). Метадані моделі (Class Meta) використовують це поле для упорядкування записів, коли вони повертаються у запиті;
- **status** - це CharField, що визначає список choice/selection. Як ви можете бачити, ми визначаємо кортеж, який містить кортежі пар ключ-значення і передаємо його аргументу вибору. Значення в key/value парі - це значення, яке користувач може вибрати, а ключі - це значення, які фактично зберігаються, якщо обрана опція. Ми також встановили значення за замовчуванням «m» (технічне обслуговування), оскільки книги спочатку будуть створені недоступними, перш ніж вони зберігатимуться на полицях.

Модель __str __ () представляє об'єкт **BookInstance**, використовуючи комбінацію його унікального ідентифікатора та пов'язаного з ним заголовка книги.

Модель автора:

Скопіюйте модель автора під існуючим кодом у **models.py**:

```
class Author(models.Model):

    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    date_of_birth = models.DateField(null=True, blank=True)
    date_of_death = models.DateField('Died', null=True,
blank=True)

    def get_absolute_url(self):
        return reverse('author-detail', args=[str(self.id)])


    def __str__(self):
        return '%s, %s' % (self.last_name, self.first_name)
```

Тепер усі поля/методи мають бути знайомі. Модель визначає автора як ім'я, прізвище, дату народження та (необов'язкову) дату смерті. Він вказує, що за умовчанням __str __ () повертає ім'я у прізвище, порядковий номер першого імені. Метод **get_absolute_url()** скасовує зіставлення URL-адреси автора з метою отримання URL-адреси для відображення окремого автора.

Тепер всі моделі створені. Перевстановіть міграцію бази даних, щоб додати їх до бази даних.

```
python3 manage.py makemigrations
python3 manage.py migrate
```

Тема 13. МАРШРУТИЗАЦІЯ ТА ОБРОБКА URL ЗАПИТІВ. CRUD. REST

Найважливішою частиною будь-якого вебфреймворку є механізм, який відповідає за маршрутизацію. Наприклад, у *Flask* для побудови карти маршрутів використовувалися спеціальні декоратори. У *Django* для цього використовується свій невеликий **eDSL**. Він описує **urlpatterns** – набір зразків, з якими зіставляються шляхи з кожного запиту.

Кожен зразок складається з опису статичних та динамічних частин шляху у вигляді рядка або регулярного виразу:

- Статичні частини шляху у зразку просто перевіряються на рівність відповідним ділянкам шляху у запиті;
- Динамічні ділянки шляху дозволяють захоплювати значення та передавати у **view** як аргументи.

Як тільки з'ясовується, що шлях або його початок співпали зі зразком, відбувається або виклик **view**, або передача частини шляху, що залишилася, у вкладений блок **urlpatterns**. У більшості великих Django-проектів **urlpatterns** вкладені один в одного і є деревом.

У цьому уроці детально розберемо статичні та динамічні маршрути, а також розглянемо вкладені urlpatterns та зворотні маршрути.

Статичні маршрути

Опишемо наступний статичний маршрут:

```
urlpatterns = [
    path('', views.index),
]
```

Тут path зіставляє зразок ' ' з **views.index**. Зразок «порожній рядок» відповідає порожньому шляху запитам головної сторінки сайту. Будь-який не порожній шлях не

збігатиметься з таким зразком. Статичні зразки зазвичай описуються рядками виду **'fruits/apples/golden_one** і чекають на запити строго цим же шляхом.

Ім'я домену не фігурує в **urlpatterns**, що дозволяє розміщувати один і той же додаток на будь-якому домені.

Динамічні маршрути

Розробники Django – прихильники використання URL, що читаються. Це означає, що маршрути в програмах Django виглядають так, що зрозуміло, куди веде шлях. Наприклад, у шляху */users/42/pets/101/med_info/* можна здогадатися, що запитується медична інформація (*med_info*) для тварини з ідентифікатором 101 (*pets/101*). Він належить користувачеві з ідентифікатором 42 (*user/42*).

Іноді виходить піти далі і замість ідентифікаторів використати імена. Наприклад, таке можливе для імен користувачів, які зазвичай є унікальними в межах системи. URL-адреса під час використання імен може виглядати так: */users/~bob/books/*.

Шляхи, які включають дані — ідентифікатори та імена — називаються динамічними. І динамічні маршрути використовуються якраз із такими шляхами.

У прикладі, який визначає динамічний маршрут, вказуються іменовані динамічні ділянки. Кожна така ділянка обробляє свою частину шляху і визначає значення аргументу, який буде переданий у **view**. У результаті від view вже не потрібно будь-яка обробка шляху, хоча це і можливо.

Опишемо **urlpatterns** для прикладу шляху, який наведено вище:

```
## urls.py

urlpatterns = [
```

```
    path('users/<int:user_id>/pets/<int:pet_id>/med_info/',
med_info_view),
    …
]
## views.py

def med_info_view(request, user_id, pet_id):
    ...
```

Тут *<int:XXX>* означає ту саму динамічну частину шляху. *int* означає, що на цій ділянці шляху очікується ціле число у вигляді рядка. Якщо сервер отримає запит на шляху */users/42/pets/101/med_info/*, маршрутизація закінчиться викликом в'ю *med_info_view(request, user_id=42, pet_id=101)*.

Крім int Django надає й інші перетворювачі шляхів – **path converters**. Понад те, можна визначати і власні. А якщо шляхи специфічні, завжди можна використовувати регулярні висловлювання, щоб виділити цікаві нам частини шляху.

Вкладені urlpatterns

Іноді маршрутів стає занадто багато і серед них намічаються групи, які мають загальну статичну частину. Наприклад, це маршрути до **views** однієї програми. У цьому випадку варто скористатися можливістю включення одних **urlpatterns** до інших.

Припустимо, у нас у проекті є програма **project.users**, в якій всі views знаходяться під загальним префіксом */users/*. Нам достатньо створити модуль **project.users.urls** з описом **urlpatterns** вже без префікса і підключити модуль до кореневої **project.urls**:

```
# project.users.urls
from django.urls import path

from project.users import views
```

```
urlpatterns = [
    path('', views.users_view),
    path('<int:user_id>/pets/<int:pet_id>/med_info/',
views.pet_med_info_view),
    …
]
# project.urls
from django.urls import path, include

urlpatterns = [
    …
    path('users/', include('project.users.urls')),
    …
]
```

У новому наборі **urlpatterns** у зразків немає префікса **users**. А в основному urlpatterns зазначено, що всі шляхи, які починаються з users, потрібно зіставляти зі зразками **project.users.urls**.

Ми підключили вкладені urlpatterns за допомогою функції **django.urls.include** та вказали модуль у вигляді рядка. Можна імпортувати модуль і вказати замість мети маршруту одразу його: *path('users', project.users.urls)* — ці два варіанти еквівалентні. Але неявне підключення замість імпорту вирішує одне важливе завдання: позбавляє потенційних циклічних імпортів.

Раніше ми закоментували у нашому міні-проекті рядок *path('admin', admin.site.urls)*. Це також включення адмінки до нашої карти маршрутів за префіксом admin. Подібно до програми часто підключаються сторонні пакети, у яких власні маршрути.

Зворотні маршрути або reverse

Часто потрібно отримати правильний шлях для певного маршруту. Наприклад, необхідно комусь дати посилання на медичну картку тварини користувача. Якщо

ми вручну збиратимемо шлях із рядків, то при змінах у маршруті новий шлях може стати некоректним.

Щоб була можливість для будь-якого маршруту завжди отримати правильний шлях, потрібно зробити операцію, зворотну маршрутизації - Django має функції **reverse** і **reverse_lazy**. Вони дозволяють отримати шлях на ім'я маршруту. Тому маршрути, які потрібно звертати, необхідно назвати (задати унікальне ім'я):

```
urlpatterns = [
    …
    path(
        '<int:user_id>/pets/<int:pet_id>/med_info/',
        views.pet_med_info_view,
        name='pet_med_info',  # <---
    ),
    …
]
```

Коли маршрут названий, можна отримати шлях викликом виду **reverse('pet_med_info', kwargs={'user_id': 42, 'pet_id': 101})**. Як би не змінювалася маршрутизація надалі, поки шлях містить ті ж іменовані ділянки і названий по-старому, ця функція даватиме актуальний для маршруту шлях.

Визначення маршрутів та функції path та re_path

Вище розглядалося зіставлення адрес URL та функцій, які обробляють запити на ці адреси. Наприклад, ми маємо такі функції у файлі **views.py**:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("<h2>Головна</h2>")
def about(request):
    return HttpResponse("<h2>Про сайт</h2>")
def contact(request):
    return HttpResponse("<h2>Контакти</h2>")
```

Це так звані функції-представлення або **view function**. І у файлі **urls.py** проекту вони порівнюються з адресами URL за допомогою функції **path()**:

```
from django.urls import path
from hello import views

urlpatterns = [
    path('', views.index),
    path('about', views.about),
    path('contact', views.contact),
]
```

За зіставлення шляхів та функцій-уявлень відповідає функція **path()**, яка розташовується в пакеті **django.urls** і яка приймає чотири параметри:

```
path(route, view, kwargs=None, name=None)
```

- **route**: представляє шаблон адреси URL, якій має відповідати запит;
- **view**: функція-подання, яке обробляє запит;
- **kwargs**: додаткові аргументи, які передаються у функцію-подання;
- **name**: назва маршруту.

```python
from django.http import HttpResponse

def index(request):
    return HttpResponse("Головна")

def about(request):
    return HttpResponse("Про сайт ")

def contact(request):
    return HttpResponse("Контакти ")
```

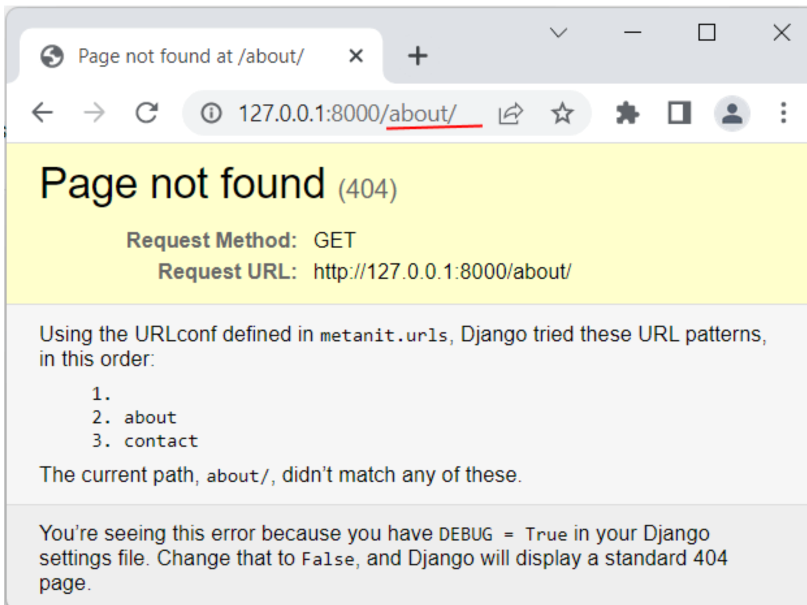
```python
from django.urls import path
from hello import views

urlpatterns = [
    path('', views.index),
    path('about', views.about),
    path('contact', views.contact),
]
```

У прикладі вище застосовувалися лише перші два параметри, які є обов'язковими: запитана URL-адреса та функція, яка обробляє запит за цією адресою. Додатково через третій параметр можна вказати ім'я маршруту:

path('', views.index, name='home'),

В данном случае маршрут будет называться "**home**".

Хоча ми можемо успішно застосовувати функцію **path()** для визначення маршрутів, вона досить обмежена за своєю дією. Запрошений шлях повинен точно відповідати вказаній в маршруті адресі URL. Так, у прикладі вище, щоб функція **views.about** могла обробляти запит, адреса повинна бути в точності "**about**". Наприклад, варто нам вказати сліш наприкінці: **"about/"** і django вже не зможе зіставити шлях із запитом.



В якості альтернативи для визначення маршрутів ми можемо використовувати функцію **re_path()**, яка також знаходиться в пакеті **django.urls** і має той самий набір параметрів:

re_path(route, view, kwargs=None, name=None)

Її переважно полягає в тому, що вона дозволяє задати адреси URL за допомогою регулярних виразів.

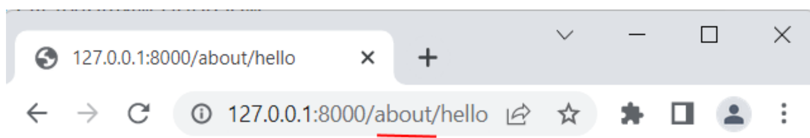
Наприклад, змінимо файл **urls.py** наступним чином:

```
from django.urls import path, re_path
from hello import views

urlpatterns = [
    path('', views.index),
    re_path(r'^about', views.about),
    re_path(r'^contact', views.contact),
]
```

Адреса в першому маршруті, як і раніше, утворюється за допомогою функції **path** і вказує на корінь вебпрограми.

Інші два маршрути утворюються за допомогою функції **re_path()**. Причому оскільки визначається регуляне вираз, то перед рядком з шаблоном адреси URL ставиться буква r. У шаблоні адреси можна використовувати різні елементи синтаксису регулярних виразів. Зокрема, вираз **^about** вказує, що адреса має починатися з "**about**". Однак він необов'язково точно повинен відповідати рядку "**about**", як це було у випадку з функцією **path**.

Наприклад, ми можемо звернутися за будь-якою адресою, головне, щоб вона починалася з "**about**", і тоді подібний запит буде оброблятися функцією **views.about**.



Послідовність маршрутів

Коли запит приходить до додатку, система перевіряє відповідність запиту маршрутам у міру їх визначення: спочатку порівнюється перший маршрут, якщо він не підходить, то порівнюється другий і так далі. Тому загальніші маршрути повинні визначатися в останню чергу, а конкретніші маршрути повинні йти на початку. Наприклад:

```
from django.urls import path, re_path
from hello import views

urlpatterns = [
    re_path(r'^about/contact/', views.contact),
    re_path(r'^about', views.about),
    path('', views.index),
]
```

Адресу "**^about/contact**" представляє більш конкретний маршрут порівняно з "**^about**". Тому він визначається в першу чергу.

Якби було навпаки, то запит на адресу "**about/contact**" оброблявся б функцією **views.about:**

```
urlpatterns = [
    path('', views.index),
    re_path(r'^about', views.about),
    re_path(r'^about/contact', views.contact),
]
```

Передача значень у функцію

Вище були розглянуті всі параметри функцій **path** і **re_path**, крім одного – **kwargs**, що дозволяє передати у функцію-представлення деякі значення. Наприклад, у файлі **views.py** визначимо такі функції:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("<h2>Главная</h2>")

def about(request, name, age):
    return HttpResponse(f"""
            <h2>О пользователе</h2>
            <p>Имя: {name}</p>
            <p>Возраст: {age}</p>
    """)
```

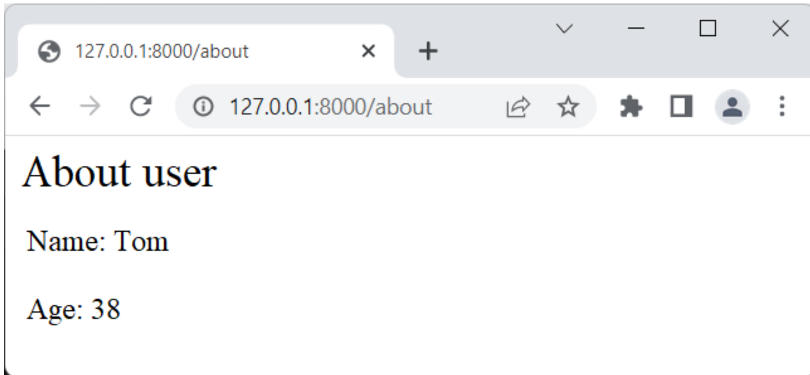
Тут функція **about()** також приймає два додаткові параметри: **name** і **age** (умовно ім'я та вік користувача). У функції їх значення надсилаються користувачеві разом із рештою вмісту.

Змінимо файл **urls.py**:

```
from django.urls import path
from hello import views

urlpatterns = [
```

```
        path('', views.index),
        path('about', views.about, kwargs={"name":"Tom", "age": 38}),
    ]
```

За допомогою параметра **kwargs** у функцію **about** передається словник із двома значеннями - для двох параметрів функції. Відповідно при зверненні до цієї функції ми побачимо у браузері відповідні дані:



CRUD-операції

Працюючи з моделями та базами даних є 4 базові операції: створення (**create**), читання (**read**), редагування (**update**) і видалення (**delete**). Всі 4 базові функції називають коротко **CRUD** - за першими літерами кожної операції.

Розглянемо базові операції із моделями на найпростішому прикладі. створення та виведення об'єктів моделі на прикладі. Нехай у файлі **models.py** визначено модель **Person**:

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=20)
    age = models.IntegerField()
```

У файлі **views.py** пропишемо чотири уявлення для отримання, збереження, редагування та видалення даних:

```
from django.shortcuts import render
from django.http import HttpResponseRedirect,
HttpResponseNotFound
from .models import Person

# отримання даних з бд
def index(request):
    people = Person.objects.all()
    return render(request, "index.html", {"people":
people})

# збереження даних в бд
def create(request):
    if request.method == "POST":
        person = Person()
        person.name = request.POST.get("name")
        person.age = request.POST.get("age")
        person.save()
    return HttpResponseRedirect("/")

# зміна даних в бд
def edit(request, id):
    try:
        person = Person.objects.get(id=id)

        if request.method == "POST":
            person.name = request.POST.get("name")
            person.age = request.POST.get("age")
            person.save()
            return HttpResponseRedirect("/")
        else:
            return render(request, "edit.html", {"person":
person})
    except Person.DoesNotExist:
```

```
            return HttpResponseNotFound("<h2>Person not found</h2>")

    # видалення даних з бд
    def delete(request, id):
        try:
            person = Person.objects.get(id=id)
            person.delete()
            return HttpResponseRedirect("/")
        except Person.DoesNotExist:
            return HttpResponseNotFound("<h2>Person not found</h2>")
```

У функції **index()** отримуємо всі дані за допомогою методу **Person.objects.all()** та передаємо їх у шаблон **index.html**.

У функції **create()** отримуємо дані із запиту типу **POST**, зберігаємо дані за допомогою методу **save()** та виконуємо переадресацію на корінь веб-сайту (тобто на функцію **index**).

Функція **edit** виконує редагування об'єкта. Функція як параметр приймає ідентифікатор об'єкта у базі даних. І спочатку з цього ідентифікатора ми намагаємося знайти об'єкт за допомогою методу **Person.objects.get(id=id)**. Оскільки у разі відсутності об'єкта ми можемо зіткнутися з винятком **Person.DoesNotExist**, то відповідно нам треба обробити подібний виняток, якщо раптом буде передано неіснуючий ідентифікатор. І якщо об'єкт не буде знайдений, користувачу повертається помилка 404 через виклик return **HttpResponseNotFound()**.

Якщо об'єкт знайдено, обробка ділиться на дві гілки. Якщо запит **POST**, тобто якщо користувач надіслав нові змінені дані для об'єкта, зберігаємо ці дані в бд і виконуємо переадресацію на корінь веб-сайту. Якщо запит **GET**, то відображаємо користувачеві сторінку edit.html із формою для редагування об'єкта.

Функція **delete** аналогічно знаходить об'єкт і виконує його видалення.

REST

Django REST framework (DRF) – це потужний та гнучкий інструмент для створення Web API на основі Django. Він надає зручні засоби для створення RESTful API, підтримує автентифікацію, авторизацію, серіалізацію, валідацію та інші функції.

Розглянемо процес створення простого застусунку з використанням цього фреймворку:

Крок 1: Першим кроком є встановлення Django REST framework. Ви можете встановити його за допомогою *pip*:

```
pip install djangorestframework
```

Крок 2: Для створення проекту Django використовуйте команду:

```
django-admin startproject myproject
```

Крок 3: Створіть програму Django за допомогою команди:

```
python manage.py startapp myapp
```

Крок 4: Налаштування Django REST framework

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'myapp',
]
```

Додайте REST framework middleware у MIDDLEWARE у файлі settings.py:

```
MIDDLEWARE = [
    ...
'rest_framework.middleware.AuthenticationMiddleware',
```

'rest_framework.middleware.AuthorizationMiddleware',
]

Крок 5: Створіть модель Django у файлі **models.py**:

```
django.db import models
class Product(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
    price = models.DecimalField(max_digits=10,
decimal_places=2)
    created_at =
models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    def __str__(self):
        return self.name
```

Крок 6: Визначте серіалізатор Django REST framework у файлі **serializers.py**:

```
rest_framework import serializers
from myapp.models import Product
class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = ['id', 'name', 'description', 'price',
'created_at', 'updated_at']
```

Крок 7: Внесіть виклик **Django REST framework** у **views.py**:

```
rest_framework import generics
from myapp.models import Product
from myapp.serializers import ProductSerializer
class ProductList(generics.ListCreateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

```
class
ProductDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Product.objects.all()
    serializer_class = Product
```

Крок 8: Визначте маршрути Django REST framework у файлі **urls.py**:

```
django.urls import path
from myapp.views import ProductList, ProductDetail
urlpatterns = [
    path('products/', ProductList.as_view(),
name='product-list'),
    path('products/<int:pk>/', ProductDetail.as_view(),
name='product-detail'),
]
```

Крок 9: Запустіть Django за допомогою команди:

```
python manage.py runserver
```

Крок 10: Тестування **Web API**

Відкрийте веб-браузер та перейдіть за адресою http://127.0.0.1:8000/products/. Ви повинні побачити список усіх продуктів.

Щоб створити новий продукт, надішліть POST-запит на http://127.0.0.1:8000/products/ з даними у форматі JSON:

```
{
    "name": "Product 1",
    "description": "Description for Product 1",
    "price": 10.99
}
```

Щоб отримати деталі конкретного продукту, надішліть GET запит на http://127.0.0.1:8000/products/1/, де 1 - ідентифікатор продукту.

Щоб оновити продукт, надішліть PUT-запит на http://127.0.0.1:8000/products/1/ з даними у форматі JSON:

```json
{
    "name": "Updated Product 1",
    "description": "Updated Description for Product 1",
    "price": 12.99
}
```

Щоб видалити продукт, надішліть DELETE-запит на http://127.0.0.1:8000/products/1/.

Django REST framework забезпечує зручні засоби для створення RESTful API, підтримує автентифікацію, авторизацію, серіалізацію, валідацію та інші функції. З його допомогою ви можете швидко та легко створювати потужні та гнучкі Web API на базі Django.

Тема 14. КОНФІГУРУВАННЯ СИСТЕМИ УПРАВЛІННЯ КОНТЕНТОМ

Система управління контентом (*Django CMS, content management system*) – система, розроблена для вебзастосунків на основі фреймворку Django. Сьогодні із трьох десятків інших доступних варіантів дана система, мабуть, є найпопулярнішою.

Нижче опишемо встановлення та налаштування **Django CMS** на Debian 7 та Ubuntu 13 VPS, а також підготовку даної системи до використання.

Pip – менеджер пакетів, який допомагає встановити необхідні програмні пакети (інструменти, бібліотеки, додатки та ін.).

Бібліотека **setuptools** заснована на функціональності набору стандартних утиліт поширення програмного забезпечення мови Python під назвою **distutils**.

У Python багато завдань виконуються дуже легко, і встановлення пакетів та додатків – не виняток. Тим не менш, значна кількість цих пакетів постачається залежно від інших пакетів. При інсталяції вони стають загальносистемними – тобто будь-яка програма Python може підключатися до цих бібліотек і використовувати їх.

За певних обставин це може призвести до серйозних збоїв налаштованих і стабільно працюючих додатків. Будь-який встановлений чи віддалений пакет однак впливає всю систему; отже, неправильна версія бібліотеки або модуля може спричинити загальносистемні пошкодження. Тому часто на початковому етапі розробки необхідне чисте робоче середовище.

Саме для цього існує інструмент **virtualenv**; з його допомогою можна відокремити репозиторій програми **Django CMS** та його складні залежності від усієї системи, що одночасно допомагає підтримувати порядок у системі та полегшує технічне обслуговування.

Установка Django CMS

Процес установки складається із п'яти дій:

1. Підготовка операційної системи.

Спочатку потрібно оновити систему. Оновіть список програмного забезпечення репозиторію, а потім – встановлені інструменти до більш актуальних версій:

aptitude update

aptitude upgrade

Тепер можна приступити до встановлення інших необхідних програмних засобів та бібліотек, а саме:

- **python-dev**: цей пакет розширює установку Python за промовчанням на систему;
- **libjpeg-dev/libpng-dev**: ці бібліотеки будуть необхідні для обробки зображень із PIL;
- **libpq-dev**: версія libpq (PostgreSQL), яка знадобиться під час розробки.

Щоб завантажити та встановити їх, введіть наступну команду:

aptitude install libpq-dev python-dev libjpeg-dev libpng-dev

2: Встановлення віртуального середовища.

Усі необхідні пакети працюють на Python. Установка Debian 7 за замовчуванням постачається з версією Python 2.7+, яка відповідає вимогам розробки. Тому можна перейти до інсталяції **pip**, який знадобиться для установки **virtualenv** (та інших пакетів).

Перш ніж встановити **pip**, необхідно встановити його залежність – **setuptools**.

Завантажити інсталяційні файли **setuptools** можна за допомогою інструмента під назвою **curl**. Дані файли не тільки автоматизують процес встановлення, але й нададуть останню версію потрібної програми. На даному етапі curl перевірить SSL сертифікати з вихідного коду і передасть дані інтерпретатору Python.

Виконайте таку команду:

$ curl https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py | python

Це дозволить встановити setuptools в рамках всієї системи.

Тепер можна встановити та налаштувати pip.

Для завантаження та встановлення інструменту знову скористайтеся **curl**. Запустіть команду:

$ curl https://raw.github.com/pypa/pip/master/contrib/get-pip.py | python -

За замовчуванням pip встановлює файли **/usr/local/bin**. Цей шлях потрібно внести до PATH, щоб мати можливість запускати цей інструмент за допомогою команди pip. Отже, виконайте:

export PATH="/usr/local/bin:$PATH"

Після інсталяції менеджера пакетів pip установка решти пакетів зводиться до одного рядку: **pip install** *имя_пакета*. Тим не менш, щоб отримати останню версію virtualenv, потрібно вказати **pip** її адресу.

Щоб pip встановив **virtualenv**, виконайте команду:

pip install https://github.com/pypa/virtualenv/tarball/1.9.X

Стандартна установка virtualenv має такий вигляд:

pip install virtualenv

Це також встановить **virtualenv** в рамках всієї системи.

3: Підготовка віртуального середовища (Venv) для Django CMS.

Тепер усі необхідні інструменти готові, можна переходити до підготовки віртуального середовища для зберігання **Django CMS**.

Для початку потрібно ініціювати **venv** (virtual environment) під назвою django_cms за допомогою virtualenv інструменту і перейти в папку проекту:

```
virtualenv django_cms
cd django_cms
```

У цьому випадку ім'я папки репозиторію проекту – **django_cms**, але можна, звичайно, встановити будь-яке ім'я. Майте на увазі, що ім'я, ніяк не пов'язане з проектом, в майбутньому може призвести до проблем з обслуговуванням.

Для використання віртуального середовища необхідно його активувати:

```
source bin/activate
```

Для дезактивації середовища використовується команда **deactivate**.

4. Встановлення залежностей Django CMS.

Встановлення pillow (заміна PIL)

Однією з необхідних залежностей є бібліотека **Python Imaging Library** (або PIL), яка використовується Django CMS (разом з іншими бібліотеками, встановленими раніше) для обробки зображень.

Але замість PIL іноді краще використовувати зручніше відгалуження цієї бібліотеки під назвою **Pillow**. Цей пакет сумісний з setuptools і автоматично усуває деякі проблеми, пов'язані з використанням PIL у віртуальному середовищі.

Щоб скачати та встановити Pillow, виконайте команду:

```
django_cms$ pip install pillow
```

Оскільки віртуальне середовище було активоване, Pillow не буде встановлено загальносистемно.

Встановлення драйверів бази даних

Django CMS дозволяє вибрати кілька процесорів бази даних для живлення програми, а саме PostgreSQL, MySQL, Oracle та SQLite. Розробники Django рекомендують

використовувати **PostgreSQL** (для цього необхідно встановити деякі бібліотеки та драйвера, які дозволять використовувати **PostgreSQL** як внутрішній інтерфейс програми).

Адаптер бази даних PostgreSQL, який використовує Django, називається **psycopg2**. Для його роботи потрібна бібліотека **libpq-dev**. Тому можна виконати наступну команду для встановлення **psycopg2** у venv:

django_cms$ pip install psycopg2

Далі використовується база даних SQLite. Для подальшої роботи з PostgreSQL, будь ласка, встановіть цей параметр.

5. Встановлення та налаштування Django CMS всередині віртуального середовища Python.

Встановлення Django CMS.

Django CMS поставляється з низкою інших компонентів, які також потрібно встановити. Тим не менш, завдяки pip їх можна встановити і налаштувати автоматично за допомогою пакету Django CMS: **django-cms**.

Для завершення встановлення запустіть наступну команду:

django_cms$ pip install django-cms

Тепер всі необхідні компоненти встановлені: Django, django-classy-tags, south, html5lib, django-mptt, django-sekizai.

Налаштування Django CMS.

Створення проекту Django CMS складається із двох етапів. Спочатку потрібно створити звичайний проект Django у віртуальному середовищі, а потім перейти до його налаштування, щоб він працював як проект Django CMS.

Отже, створіть проект Django. Назвемо його dcms:

django_cms$ django-admin.py startproject dcms

django_cms$ cd dcms

Щоб перевірити інсталяцію, перш ніж перейти до налаштування проекту, запустіть наступне (це дія запустить простий сервер розробки, до якого можна отримати доступ ззовні):

django_cms$ python manage.py runserver 0.0.0.0:8000

Тепер введіть URL-адресу в браузері, замінюючи 0.0.0.0 реальною IP-адресою.

Тема 15. ТЕСТУВАННЯ ТА НАЛАГОДЖЕННЯ DJANGO ПРОЕКТІВ. ФРЕЙМВОРК PYTEST

Сайти під час проектування та розробки стає дедалі складніше тестувати вручну. Також дедалі складнішими стають внутрішні взаємодії між компонентами – внесення невеликої зміни в одній частині програми впливає на інші. При цьому, щоб усе продовжувало працювати, потрібно вносити все більше і більше змін так, щоб не додавалися нові помилки. Одним із способів, який дозволяє пом'якшити наслідки додавання змін, є впровадження в розробку автоматичного тестування – воно повинно просто і надійно запускатися щоразу, коли ви вносите зміни до свого коду. Розглнемо питання автоматизації юніт-тестування сайту за допомогою фреймворку Django на прикладі прототипу бібліотеки.

LocalLibrary містить сторінки для показу списків всіх книг, авторів, детальної інформації про книги **Book** та авторів **Author**, а також сторінку для оновлення інформації про екземплярі книги **BookInstance** і, крім того, сторінки для створення, оновлення та видалення записів моделі **Author**. Навіть у разі невеликого сайту, ручний перехід на кожну сторінку та перевірка того, що все працює як слід, може зайняти кілька хвилин. У процесі внесення змін та зростання сайту потрібний час для проведення перевірок лише зростатиме.

Автоматичні тести можуть серйозно допомогти нам упоратися з цією проблемою. Очевидними перевагами в такому випадку є значно менші часові витрати на проведення тестів, їх докладне виконання, а крім того, тести мають постійну функціональність або послідовність дій. У зв'язку зі швидкістю їх виконання автоматичні тести можна виконувати більш часто, а якщо вони проваляться, то вкажуть на відповідне місце.

Крім того, автоматичні тести можуть діяти як перший "справжній користувач" вашого коду, змушуючи вас

стежити за оголошеннями та документуванням поведінки вашого сайту. Тести часто є основою для створення прикладів вашого коду та документації. З цих причин іноді деякі процеси розробки програмного забезпечення починаються з визначення тестів та їх реалізації, а вже після цього слідує написання коду, який повинен мати відповідну поведінку (так звана розробка на основі тестів та на основі поведінки).

Існує кілька типів, рівнів, класифікацій тестів та тестових прийомів. Найбільш важливими автоматичними тестами є:

- **Юніт-тести**. Перевіряють функціональну поведінку окремих компонентів, часто класів і функций;
- **Регресійне тестування**. Тести, які відтворюють помилки (баги). Кожен тест спочатку запускається для перевірки того, що баг був виправлений, а потім перезапускається для того, щоб переконатися, що він не був внесений знову з появою нових змін коду;
- **Інтеграційні тести**. Перевірка співпраці груп компонентів. Ці тести відповідають за спільну роботу між компонентами, не звертаючи уваги на внутрішні процеси в компонентах. Вони проводяться як для простих груп компонентів, так і для вебсайтів.

Тестування сайту це складне завдання, тому що воно складається з кількох логічних шарів – від HTTP-запиту та запиту до моделей, до валідації форми та їх обробки, а крім того, рендерингу шаблонів сторінок.

Django пропонує фреймворк для створення тестів, побудованого на основі ієрархії класів, які, в свою чергу, залежать від стандартної бібліотеки **Python unittest**.

Незважаючи на назву, цей фреймворк підходить і для юніт-, і для інтеграційного тестування. Фреймворк Django додає методи API та інструменти, які допомагають тестувати як веб так і специфічну для Django поведінку. Це дозволяє вам імітувати URL-запити, додавання тестових даних, а також проводити перевірку вихідних даних ваших програм. Крім того, Django надає API (**LiveServerTestCase**) та інструменти для застосування різних фреймворків тестування, наприклад, ви можете підключити популярний фреймворк **Selenium** (en-US) для імітації поведінки користувача в реальному браузері.

Для написання тесту ви повинні успадковуватися від будь-якого з класів тестування Django (або юніттесту) (**SimpleTestCase, TransactionTestCase, TestCase, LiveServerTestCase**), а потім реалізувати окремі методи перевірки коду (тести це функції-"затвердження", які перевіряють, що результатом True або False, або що два значення рівні тощо). Коли ви запускаєте тест, фреймворк виконує відповідні тестові методи у вашому спадковому класі. Методи тестування запускаються незалежно один від одного, починаючи з методу налаштувань та/або завершуючись методом руйнування (tear-down), визначеним у класі, як показано нижче.

```
class YourTestClass(TestCase):

def setUp(self):
    # налаштування перед кожним тестом
    pass

def tearDown(self):
    # очистка після кожного методу
    pass

def test_something_that_will_pass(self):
    self.assertFalse(False)
```

```
def test_something_that_will_fail(self):
        self.assertTrue(False)
```

Найкращий базовий клас для більшості тестів це **django.test.TestCase**. Цей клас створює чисту базу даних перед запуском своїх методів, а також запускає кожну функцію тестування у власній транзакції. Цей клас також має тестовий **Клієнт**, який ви можете використовувати для імітації взаємодії користувача з кодом на рівні відображення. У наступних розділах ми зосередимося на юніт-тестах, які будуть створені на основі класу **TestCase**.

Загалом, варто перевіряти поведінку власного коду, і не варто перевіряти працездатність Django чи сторонніх бібліотек.

Наприклад, розглянемо модель **Author**, визначену нижче. Вам не потрібно перевіряти той факт, що **first_name** і **last_name** були збережені в базі даних як **CharField**, тому що за це відповідає безпосередньо Django (хоча звичайно, на практиці протягом розробки ви опосередковано перевірятимете цю функціональність). Теж стосується і, наприклад, перевірки того, що поле date_of_birth є датою, оскільки це також частина реалізації Django.

Ви повинні перевірити текст для позначок (First name, Last_name, Date of birth, Died), і розмір поля, виділеного для тексту (100 символів), тому що вони є частиною вашої розробки і чимось, що може зламатися/змінитися в майбутньому.

```
class Author(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    date_of_birth = models.DateField(null=True,
blank=True)
    date_of_death = models.DateField('Died', null=True,
blank=True)

    def get_absolute_url(self):
```

```
        return reverse('author-detail', args=[str(self.id)])

    def __str__(self):
        return '%s, %s' % (self.last_name, self.first_name)
```

Подібним чином ви повинні переконатися, що методи **get_absolute_url()** і **__str__()** поводяться як потрібно, тому що вони є частиною вашої бізнес логіки. У випадку функції **get_absolute_url()** ви можете бути впевнені, що функція з Django **reverse()** була реалізована правильно і, отже, ви тестуєте лише те, щоб відповідний виклик відображено правильно.

Django використовує юніт-тестовий модуль – вбудований "виявник" тестів, який знаходить тести у поточній робочій директорії, у будь-якому файлі з шаблонним ім'ям **test*.py**. Надаючи відповідні імена файлів, ви можете працювати з будь-якою структурою, яка вас влаштовує. Ми рекомендуємо створити пакет для вашого тестуючого коду і, отже, відокремити файли моделей, відображень, форм та будь-які інші, від коду, який буде використовуватися для тестів. Наприклад:

```
catalog/
  /tests/
    __init__.py
    test_models.py
    test_forms.py
    test_views.py
```

У проекті **LocalLibrary** створіть файлову структуру, вказану вище. Файл **__init__.py** має бути порожнім (так ми кажемо Пітону, що ця директорія є пакетом). Ви можете створити три тестові файли за допомогою копіювання та перейменування файлу-зразка **/catalog/tests.py**.

Відкрийте **/catalog/tests/test_models.py.** Файл повинен імпортувати **django.test.TestCase**, як показано нижче:

```
from django.test import TestCase
```

Ви часто додаватимете відповідний тестовий клас для кожної моделі/відображення/форми з окремими методами перевірки кожної окремої функціональності. У деяких випадках ви захочете мати окремий клас для тестування якогось особливого варіанта роботи, або функціональності, з окремими функціями тестування, які перевірятимуть елемент/елементи даного варіанту (наприклад, ми можемо створити окремий клас тестування для перевірки того, що поле валідно, – функції даного класу перевірятимуть кожен неправильний варіант використання). Знову ж таки, структура файлів і пакетів повністю залежить від вас і буде краще, якщо ви її дотримуватиметеся.

Додайте тестовий клас, показаний нижче, до нижньої частини файлу. Цей клас демонструє, як створити клас тестування за допомогою успадкування від **TestCase**.

```
class YourTestClass(TestCase):

    @classmethod
    def setUpTestData(cls):
        print("setUpTestData: Run once to set up non-
modified data for all class methods.")
        pass

    def setUp(self):
        print("setUp: Run once for every test method to
setup clean data.")
        pass

    def test_false_is_false(self):
        print("Method: test_false_is_false.")
        self.assertFalse(False)

    def test_false_is_true(self):
        print("Method: test_false_is_true.")
        self.assertTrue(False)
```

```
    def test_one_plus_one_equals_two(self):
        print("Method: test_one_plus_one_equals_two.")
        self.assertEqual(1 + 1, 2)
```

Цей клас визначає два методи, які ви можете використовувати для дотестового налаштування (наприклад, створення будь-якої моделі, або інших об'єктів, які вам знадобляться):

- **setUpTestData()** викликається щоразу перед запуском тесту лише на рівні налаштування всього класу. Ви повинні використовувати цей метод для створення об'єктів, які не будуть модифікуватися/змінюватися в жодному з тестових методів;
- **setUp()** викликається перед кожною функцією тестової для налаштування об'єктів, які можуть змінюватися під час тестів (кожна функція тестування буде отримувати "свіжу" версію даних об'єктів).

Далі йдуть кілька методів, які використовують функції **Assert**, що перевіряють умови "істинно" (true), "хибно" (false) або рівність (**AssertTrue, AssertFalse, AssertEqual**). Якщо умови не виконуються як очікувалося, це призводить до провалу тесту і висновку відповідного повідомлення про помилку на консоль.

Функції перевірки тверджень **AssertTrue, AssertFalse, AssertEqual** реалізовані в **unittest**. У цьому фреймворку існують і інші подібні функції, а крім того, специфічні для Django функції перевірки, наприклад, переходу з/до відображення (**assertRedirects**), перевірки використання якогось конкретного шаблону (**assertTemplateUsed**) тощо.

Найпростішим способом запуску всіх тестів є застосування наступної команди:

```
python3 manage.py test
```

Таким чином ми знайдемо в поточній директорії всі файли з ім'ям **test*.py** і запустимо всі тести (у нас є кілька

файлів для тестування, але на даний момент тільки **/catalog/tests/test_models.py містить тести**). За замовчуванням тести повідомлять що-небудь, тільки у разі провалу.

Запустіть тести із кореневої папки сайту **LocalLibrary**. Ви повинні побачити текст, який нагадує наступний:

```
>python manage.py test

Creating test database for alias 'default'...
setUpTestData: Run once to set up non-modified data for all class methods.
setUp: Run once for every test method to setup clean data.
Method: test_false_is_false.
.setUp: Run once for every test method to setup clean data.
Method: test_false_is_true.
.setUp: Run once for every test method to setup clean data.
Method: test_one_plus_one_equals_two.
.
======================================================================
FAIL: test_false_is_true (catalog.tests.tests_models.YourTestClass)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\Github\django_tmp\library_w_t_2\locallibrary\catalog\tests\tests_models.py", line 22, in test_false_is_true
    self.assertTrue(False)
AssertionError: False is not true
```

-

Ran 3 tests in 0.075s

FAILED (failures=1)
Destroying test database for alias 'default'...

Як бачите, один тест провалився і ми можемо точно побачити, в якій саме функції це сталося і чому (так і було задумано, оскільки False не дорівнює True!).

Для конкретного прикладу, далі розглянемо тестування моделей нашого проекту.

Розглянемо модель **Author**. Ми повинні провести тести текстових міток усіх полів, оскільки, незважаючи на те, що не всі вони визначені, у нас є проект, в якому сказано, що всі їх значення мають бути задані. Якщо ми не проведемо їх тестування, тоді ми не знатимемо, що ці мітки дійсно містять необхідні значення. Ми впевнені, що Django створить поле заданої довжини, таким чином наші тести перевірятимуть потрібний нам розмір поля, а заразом і його вміст.

```
class Author(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    date_of_birth = models.DateField(null=True,
blank=True)
    date_of_death = models.DateField('Died', null=True,
blank=True)

    def get_absolute_url(self):
        return reverse('author-detail', args=[str(self.id)])

    def __str__(self):
        return '%s, %s' % (self.last_name, self.first_name)
```

Відкрийте файл **/catalog/tests/test_models.py** і замініть його вміст кодом, наведеним у фрагменті для

тестування моделі **Author** (фрагмент представлений нижче).

У першому рядку ми імпортуємо клас **TestCase**, а потім успадковуємося від нього, створюючи клас з описовим ім'ям (**AuthorModelTest**), воно допоможе нам ідентифікувати місця помилок у тестах під час виведення інформації на консоль. Потім ми створюємо метод **setUpTestData()**, в якому створюємо об'єкт автора, який ми будемо використовувати у тестах, але ніде не змінюватимемо.

```
from django.test import TestCase

# Create your tests here.
from catalog.models import Author
class AuthorModelTest(TestCase):
    @classmethod
    def setUpTestData(cls):
        #Set up non-modified objects used by all test
methods
        Author.objects.create(first_name='Big',
last_name='Bob')

    def test_first_name_label(self):
        author=Author.objects.get(id=1)
        field_label =
author._meta.get_field('first_name').verbose_name
        self.assertEquals(field_label,'first name')

    def test_date_of_death_label(self):
        author=Author.objects.get(id=1)
        field_label =
author._meta.get_field('date_of_death').verbose_name
        self.assertEquals(field_label,'died')

    def test_first_name_max_length(self):
        author=Author.objects.get(id=1)
```

```
            max_length = 
author._meta.get_field('first_name').max_length
            self.assertEquals(max_length,100)

        def 
test_object_name_is_last_name_comma_first_name(self):
            author=Author.objects.get(id=1)
            expected_object_name = '%s, %s' % 
(author.last_name, author.first_name)
            self.assertEquals(expected_object_name,str(author))

        def test_get_absolute_url(self):
            author=Author.objects.get(id=1)
            #This will also fail if the urlconf is not defined.

self.assertEquals(author.get_absolute_url(),'/catalog/author/1')
```

Тести полів перевіряють значення текстових міток (**verbose_name**), включаючи їхню очікувану довжину. Усі методи мають описові імена, які логіка дотримується однієї й тієї структури:

```
author=Author.objects.get(id=1)

field_label = 
author._meta.get_field('first_name').verbose_name

self.assertEquals(field_label,'first name')
```

Цікаво відзначити наступне:

- Ми не можемо отримати поле **verbose_name** безпосередньо через **author.first_name.verbose_name**, тому що **author.first_name** є рядком. Натомість нам потрібно використовувати атрибут **_meta** об'єкта автора для отримання того екземпляра поля, який буде використовуватися для отримання додаткової інформації;

- Ми вибрали метод **assertEquals(field_label,'first name')** замість **assertTrue(field_label == 'first name')**, тому що, у разі провалу тесту, у висновку буде вказано яке саме значення містить мітка і це трохи полегшить нам завдання з налагодження коду.

Крім того, нам слід провести тести наших власних методів. Вони просто перевіряють, що імена об'єктів мають наступні значення "**Last Name, First Name**" і що URL-адреса, за якою ми отримуємо екземпляр Author, такий, як очікується.

```
def
test_object_name_is_last_name_comma_first_name(self):
    author=Author.objects.get(id=1)
    expected_object_name = '%s, %s' %
(author.last_name, author.first_name)
self.assertEquals(expected_object_name,str(author))
def test_get_absolute_url(self):
    author=Author.objects.get(id=1)

self.assertEquals(author.get_absolute_url(),'/catalog/author/1')
```

Тепер запустіть випробування. Якщо ви створили модель **Author**, відповідно до розділу про моделі цього посібника, то ймовірно, що ви отримаєте повідомлення про помилку для мітки **date_of_death**, як показано нижче. Тест провалився тому, що, відповідно до угоди Django, перший символ імені мітки має бути у верхньому регістрі (Django робить це автоматично).

```
======================================
============================
FAIL: test_date_of_death_label
(catalog.tests.test_models.AuthorModelTest)
------------------------------------------------------------
-
Traceback (most recent call last):
```

```
File "D:\...\locallibrary\catalog\tests\test_models.py", line 32, in test_date_of_death_label
    self.assertEquals(field_label,'died')
AssertionError: 'Died' != 'died'
- Died
? ^
+ died
? ^
```

Це несуттєвий баг, але він демонструє нам те, що написання тестів може ретельніше перевірити всі неточності, які ви можете зробити.

Django-фреймворк для тестування допомагає також створювати ефективні юніт- та інтеграційні тести - ми розглянули лише невелику частину того, що може робити фреймворк unittest і зовсім не згадували доповнення Django (наприклад, модуль **unittest.mock**, який підключає сторонні бібліотеки тестування).

З усієї безлічі сторонніх інструментів тестування, ми коротко опишемо можливості двох:

- **Coverage**: Це інструмент Python, який формує звіти про те, скільки коду виконується під час проведення тестів. Це корисно для уточнення ступеня покриття коду тестами;
- **Selenium (en-US):** Це фреймворк проведення автоматичного тестування в цьому браузері. Він дозволяє вам імітувати взаємодію користувача з вашим сайтом (що є наступним кроком у проведенні інтеграційних тестів).

Фреймворк PyTest

Хоча під час тестування на Python і Django можна обійтися без Pytest, цей фреймворк пропонує новий підхід для написання тестів, а саме функціональне тестування для програм і бібліотек. Нижче перераховано деякі плюси та мінуси цього фреймворку.

Плюси використання Pytest:

- Оператори **Assert** (не потрібно запам'ятовувати імена `**self.assert***`);
- Детальна інформація про помилки;
- Доповлення (явні, модульні, масштабовані);
- Додаткові функції (автоматичне використання, область дії, об'єкт запиту, вкладеність, фіналізатори тощо);
- Автоматичне виявлення тестових модулів і функцій;
- Відмітки;
- Параметризація;
- Менше шаблонного коду: просто створіть файл, напишіть функцію з assert і запустіть;
- Плагіни з понад 736+ зовнішніми плагінами та процвітаючою спільнотою;
- Може запускати пакети модульних тестів.

Мінуси використання Pytest:

- Потрібні трохи більше знань Python ніж використання юніт-тестів, як декоратори та прості генератори;
- Необхідність окремої установки модуля. Але це також може бути перевагою, тому що ви не залежите від версії Python. Якщо вам потрібні нові функції, вам просто потрібно оновити пакет pytest.

Практичні завдання

Базові особливості синтаксису.

1. Реалізувати програму, яка за заданими коефіцієнтами знаходить та виводить розв'язки квадратного рівняння.
Формат виводу:
ax^2 + bx + c = 0
x_1 = s_1
x_2 = s_2
The equation does not have real solutions
2. Реалізувати програму, яка для довільних 3-х точок на площині перевіряє чи утворюють вони трикутник. Формат виводу:
(p1(x1, y1), p2(x2, y2), p3(x3, y3)) - is a / is not a triangle
3. Реалізувати програму, яка для довільних 4-х точок на площині перевіряє чи утворюють вони опуклий чотирикутник і якщо так, то виводить тип цього чотирикутника.
Формат виводу:
(p1(x1, y1), p2(x2, y2), p3(x3, y3), p4(x4, y4)) - is a ... / is not a convex quadrangle
4. Реалізувати програму, яка виводить у консоль, створені за допомогою символу *, наступні геометричні фігури:
(а) квадрат;
(б) прямокутник;
(в) трапецію;
(г) перевернуту трапецію;
(д) паралелограм;
(е) ромб що опирається на вершину;
(ж) ромб що опирається на сторону;
(и) рівнобедрений трикутник;
(к) рівносторонній трикутник;
(л) прямокутний трикутник що опирається на вершину 90;
(м) прямокутний трикутник що опирається на катет;
(н) прямокутний трикутник що опирається на гіпотенузу;

(п) літери англійської абетки.
Формат виводу:
Rectangle:
```
*******
*     *
*     *
*******
```

Типи даних.

1. Для заданих послідовностей цілих чисел та символів реалізувати алгоритми:
(а) сортування вибором;
(б) сортування вставкою;
(в) сортування обміном;
з можливістю вибору критерію сортування.
2. Реалізувати програму, яка виконує наступні трансформації:
List --> Tuple
--> String
--> Set
Tuple --> List
--> String
--> Set
String --> List
--> Tuple
--> Set
Set --> List
--> String
--> Tuple
для довільно заданих списків, кортежів, рядків та множин і виводить результати у консоль. Приклади:
[1, 2, 3, [’f’, 6]] --> (1, 2, 3, (’f’, 6))
--> ’123f6’
--> {1, 2, 3, {’f’, 6}}

3. Реалізувати програму-редактор списків, яка реалізує базові операції над списками та дозволяє через консольний інтерфейс маніпуляції над списками.
4. Реалізувати програму-редактор словників, яка реалізує базові операції над списками та дозволяє через консольний інтерфейс маніпуляції над словниками.
5. Реалізувати програму-редактор множин, яка реалізує базові операції над списками та дозволяє через консольний інтерфейс маніпуляції над множинами.

6. Реалізувати програму, яка для заданого каталогу будує та виводить у консоль дерево, що відповідає його структурі. Елементами словника є шляхи до каталогів та файлів.
Формат виводу:
/root/
/root/1st_subfolder/
/root/1st_subfolder/1st_file.ext
...
/root/final_subfolder/
/root/final_subfolder/1st_file.ext
...
7. Реалізувати програму яка аналізує вхідний файл input.txt з англомовним текстом та визначає частоти входження усіх використаних у тексті файлу:
(а) символів;
(б) знаків пунктуації;
(в) пробілів;
(г) голосних літер;
(д) приголосних літер;
(е) слів;
(ж) імен/власних назв;
(и) слів фіксованої довжини;
(к) речень фіксованої довжини;
(л) абзаців фіксованої довжини;
та записує результати аналізу у вихідний файл output.txt.

8. Реалізувати гру в шашки з використанням консольного інтерфейсу з виводом стану
ігрового поля зображеного за допомогою довільних символів з клавіатури. Формат
виводу:

```
Turn: 0, Player: X
Player X: x points
Player Y: y points
Winner:
-------------------
|-|b|-|b|-|b|-|b|
|b|-|b|-|b|-|b|-|
|-|b|-|b|-|b|-|b|
| |-| |-| |-| |-|
|-| |-| |-| |-| |
|w|-|w|-|w|-|w|-|
|-|w|-|w|-|w|-|w|
|w|-|w|-|w|-|w|-|
-------------------
```

9. Реалізувати гру хрестики-нулики з ігровим полем розмірності $n \times n$ для:
(а) двох осіб;
(б) однієї особи (гра з комп’ютером)

<u>Структури даних.</u>

1. Реалізувати програму-редактор масивів, яка реалізує базові операції та дозволяє через консольний інтерфейс виконувати маніпуляції над масивами.
2. Реалізувати програму-редактор двобічних черг, яка реалізує базові операції та дозволяє виконувати через консольний інтерфейс маніпуляції над двобічними чергами.

3. Реалізувати програму-редактор упорядкованих списків, яка реалізує базові операції та дозволяє через консольний інтерфейс виконувати маніпуляції над упорядкованими списками.
4. Реалізувати програму-редактор черг з пріоритетами, яка реалізує базові операції та дозволяє через консольний інтерфейс виконувати маніпуляції над чергами з пріоритетами.
5. Реалізувати програму яка перетворює числові масиви у черги з пріоритетами. Приклади:
array('i', [1, 5, 2, -5, 7, 3]) --> [-5, 1, 2, 5, 7, 3]
array('d', [1, 5.9, 2.25, -5.4, 7.1]) --> [-5.4, 1, 2.25, 5.9, 7.1]
6. Реалізувати програму яка перетворює двобічні числові черги у черги з пріоритетами.
Приклади:
deque([1, 5, 2, -5, 7, 3]) --> [-5, 1, 2, 5, 7, 3]
7. Реалізувати програму яка перетворює однорівневі числові кортежі у черги з пріоритетами.
Приклади:
(1, 5, 2, -5, 7, 3) --> [-5, 1, 2, 5, 7, 3]
(1, 5.9, 2.25, -5.4, 7.1) --> [-5.4, 1, 2.25, 5.9, 7.1]
8. Реалізувати програму яка перетворює однорівневі числові множини у черги з пріоритетами.
Приклади:
{1, 5, 2, -5, 7, 3} --> [-5, 1, 2, 5, 7, 3]
{1, 5.9, 2.25, -5.4, 7.1} --> [-5.4, 1, 2.25, 5.9, 7.1]

Функції.

1. Реалізувати функцію обчислення факторіалу заданого числа.
2. Реалізувати функцію обчислення n-го елементу послідовності Фібоначчі.
3. Реалізувати функцію обчислення суми елементів послідовності Фібоначчі на заданому проміжку [a, b], де a

< b – цілі числа, що визначають номери членів послідовності.
4. Реалізувати функцію генерації списку усіх простих чисел для заданого списку цілих чисел.
5. Використовуючи функції реалізувати алгоритми:
(а) швидкого сортування;
(б) сортування злиттям;
(в) пірамідального сортування;
заданих списків цілих чисел, із можливістю вибору критерію сортування.
6. Реалізувати функції шифрування та дешифрування довільної вхідної послідовності рядків. Функція шифрування повинна аналізувати вхідну послідовність та шифрувати її за допомогою шифру Цезаря із заданим зсувом. Функція дешифрування повинна виконувати декодування зашифрованої послідовності за заданим зсувом. Зсув потрібно реалізувати у вигляді окремої функції яка певним чином розраховує та повертає відповідне ціле число.
7. Реалізувати функцію, яка за заданим ключем для довільної кількості вхідних множин генерує булеан їх:
(а) об’єднання,
(б) перетину,
(в) різниці,
(г) симетричної різниці.
Приклади:
a = {1, 5, ’4’}
b = {’4’, 1}
c = {0, 1}
compute_power_set(’u’, a, b, c)
--> {{}, {1}, {5}, {’4’}, {0}, {1, 5}, {1, ’4’}, {1, 0}, {5, ’4’}, {5, 0}, {’4’, 0}, {1, 5, ’4’}, {1, 5, 0}, {1, ’4’, 0}, {5, ’4’, 0}, {1, 5, ’4’, 0}}
compute_power_set(’i’, a, b, c)
--> {{}, {1}}
compute_power_set(’d’, a, b, c)

--> {{}, {5}}
compute_power_set('sd', a, b)
--> {{}, {5}, {0}, {5, 0}}
8. Реалізувати функцію яка аналізує довільну вхідну послідовність рядків, списків, кортежів, множин, словників, масивів та генерує лінійний список елементів заданого типу str/int/float на основі елементів вхідних параметрів функції. Вхідні списки, кортежі та словники можуть мати довільну вкладеність. Приклад:
a = [3.5, 6, ['dd', (5.7)]]
b = (2, 8, ['n'], ('fff', ('dfg43')))
c = {'a', 6, 9, 3}
d = {'1': 'a', '2': {'12': {'13': 'aaa'}}}
f = array('i', [2, 7, 34, 9, 4])
s = 'sdf8894nkjfksd7'
convert('int', a, b, c, d, f, s)
--> [6, 2, 8, 6, 9, 3, 122, 2, 7, 34, 9, 4]
convert('float', a, b, c, d, f, s)
--> [3.5, 5.7]
convert('str', a, b, c, d, f, s)
--> ['dd', 'fff', 'dfg43', 'a', 'aaa', 'sdf8894nkjfksd7']

Ітератори та генератори.

1.Реалізувати генератор кортежів парних чисел на основі лінійного числового списку.
2. Реалізувати генератор списків голосних літер англійського алфавіту на основі лінійного списку довільних рядків.
3. Реалізувати генератор словників додатних цілих чисел на основі лінійного числового списку.
4. Реалізувати генератор булеану довільної множини.
5. Реалізувати генератор множини елементів списку з довільним рівнем вкладеності.

6. Реалізувати генератор списків кортежів декартового добутку довільної кількості множин/списків/кортежів.
7. Реалізувати генератор списків комбінацій фіксованої довжини додатних цілих чисел на основі множин/списків/кортежів чисел та рядків.
8. Реалізувати генератор списків парних цілих чисел на основі довільного вхідного списку елементів, де елементи можуть мати тип: str, int, float, list, tuple, set, dict.
9. Реалізувати генератор словників рядків, літери яких знаходяться у нижньому регістрі, на основі довільного вхідного словника елементів, де значення елементів можуть мати тип: str, int, float, list, tuple, set, dict.
10. Реалізувати генератор упорядкованих за зростанням числових кортежів на основі довільного вхідного списку елементів, де елементи можуть мати тип: str, int, float, list, tuple, set, dict.
11. Реалізувати генератор словників нумерованих кортежів заданої розмірності усіх можливих комбінацій елементів довільних рядків, де елементи кортежів є цифрами у шістнадцятковій системі числення.
12. Реалізувати генератор множин непарних нумерованих кортежів усіх можливих можливих перестановок елементів довільного словника без вкладень, де непарні кортежі – це кортежі сума усіх елементів яких є непарним числом.

Об'єктно-орієнтоване програмування.

1. Використовуючи абстрактні класи, класи, класи-переліки, механізми успадкування та агрегації спроектувати та реалізувати класову ієрархію опуклих чотирикутників на площині, що визначає наступні типи чотирикутників:
(а) прямокутники,
(б) квадрати,
(в) паралелограми,

(г) ромби,
(д) трапеції,
(е) прямокутні трапеції,
(ж) рівнобічні трапеції.
Для усіх класів ієрархії реалізувати:
— Атрибут що визначає тип фігури.
— Атрибут що визначає унікальний незмінний ідентифікатор фігури в залежності від її типу.
— Незмінні атрибути що визначають вершини фігур.
— Незмінні атрибути що визначають довжини сторін фігури.
— Незмінні атрибути що визначають периметр та площу фігури.
— Незмінні атрибути що визначають довжину діагоналей фігури.
— Незмінні атрибути що визначають градусну міру кутів фігури.
— Динамічні атрибути які перевіряють виконання основних якісних властивостей фігури (наприклад рівність сторін, паралельність сторін, рівність кутів, тощо).
— Метод ініціалізації об'єктів екземплярів.
— Методи отримання координат вершин фігури.
— Методи отримання довжини сторін фігури.
— Методи отримання периметру та площі фігури.
— Методи отримання довжини діагоналей фігури.
— Методи отримання градусних мір кутів фігури.
— Метод отримання переліку підтипів фігури.
— Метод отримання переліку супертипів фігури.
— Метод перевірки належності фігури до певного типу фігур в рамках ієрархії.
— Усі можливі методи порівняння за площею та периметром фігури із будь-якої іншою фігурою ієрархії. Реалізація має враховувати усі можливі варіанти порівняння: лише за площею, лише за периметром, за площею та периметром.

— Метод перевірки перетину фігури із будь-якою іншою фігурою ієрархії.
Усі класи ієрархії повинні розміщуватися в окремих модулях, а для використання імпортуватися за допомогою інструкції import ModuleName.

Умови лабораторних робіт.

Однорівнева система. Створити графічний клієнт-форму в якому реалізувати операції CRUD:

1. Робота з XML
2. Бази даних DBMS(MySQL)

Дворівнева система клієнт- сервер. Клієнт графічна форма, на серверній частині DAO клас взаємодії з DBMS.

3. Передача за допомогою Socket
4. Передача за допомогою RMI

Багаторівнева система MOM.

5. Передача між клієнтом та сервером за допомогою MQ(ApacheMQ, RabbitMQ,....).
6. Web реалізація клієнта з використанням веб сервісів
7. Реалізувати за допомогою SOAP web services
8. Реалізувати за допомогою REST web services

Варіанти завдань лабораторних робіт

варіант 1
Предметна область Карта світу
Об'єкти Країни, Міста
Примітка Карта світу містить множину
країн для кожної країни визначено множина міст.

варіант 2
Предметна область Бібліотека
Об'єкти Автори, Книги
Примітка Книги в бібліотеці згруповані по авторам. У кожного учасника є множина книг.

варіант 3
Предметна область Відділ кадрів
Об'єкти Підрозділи, Співробітники
Примітка Є множина підрозділів підприємства. В
кожному підрозділі працює множина співробітників.

варіант 4
Предметна область Навчальний відділ
Об'єкти Групи, Студенти
Примітка Є множина навчальних груп. Кожна група включає в себе множину студентів.

варіант 5
Предметна область Автосалон
Об'єкти Виробники автомобілів,марки
Примітка Марки автомобілів згруповані по виробникам. У кожного виробника є множина марок.

варіант 6
Предметна область Агентство новин

Об'єкти Категорії новин, Новини
Примітка Новини згруповані по категоріям. У кожній категорії є множина новин.

варіант 7
Предметна область Продуктовий магазин
Об'єкти Категорія продукту, Продукт
Примітка Продукти в магазині згруповані за категоріями для кожної категорії визначено множина продуктів.

варіант 8
Предметна область Футбол
Об'єкти Команди, Гравці
Примітка Є множина футбольних команд для кожної команди визначено множина гравців.

варіант 9
Предметна область Музичний магазин
Об'єкти Виконавці, Альбоми
Примітка У музичному магазині альбоми згруповані по виконавцям для кожного виконавця задано множина альбомів.

варіант 10
Предметна область Аеропорт
Об'єкти Авіакомпанії, Рейси
Примітка Є множина авіакомпаній для кожної авіакомпанії визначені її рейси.

варіант 11
Предметна область Файлова система
Об'єкти Папки, Файли
Примітка Є множина папок для кожної папки визначено множину файлів.

варіант 12
Предметна область Розклад занять
Об'єкти Дні тижня, Заняття
Примітка Є множина днів для кожного дня визначено множина занять.

варіант 13
Предметна область Нотатки
Об'єкти Календарні дні, Заходи
Примітка Є множина днів для кожного дня визначено множина заходів.

варіант 14
Предметна область Відеомагазин
Об'єкти Жанри, Фільми
Примітка Є множина жаров для кожного жанру визначено множина фільмів.

варіант 15
Предметна область Залізниця
Об'єкти Дороги, Станції
Примітка Є множина залізних доріг. У відомстві кожної дороги знаходиться множина станцій.

варіант 16
Предметна область Склад
Об'єкти Секції, Товари
Примітка Товари на складі згруповані по секціях для кожної секції задано множина товарів.

варіант 17
Предметна область Кафедра університету
Об'єкти Викладачі, Дисципліни

Примітка На кафедрі є множина викладачів для кожного викладача задано множина дисциплін.

варіант 18
Предметна область Програмне забезпечення
Об'єкти Виробники, Програмні продукти
Примітка Програмні продукти згруповані по виробникам для кожного виробника задано множина продуктів.

варіант 19
Предметна область Геометрія
Об'єкти Багатокутники, Вершини
Примітка Є множина багатокутників кожен багатокутник складається з довільного числа вершин.

варіант 20
Предметна область Схема метро
Об'єкти Лінії, Станції
Примітка Є множина ліній метрополітену кожна лінія складається з послідовності станцій.

Організація оцінювання:

Терміни проведення форм оцінювання:

1. Контрольна робота 1 (тест): до 7 тижня семестру.

2. Контрольна робота 2 (тест): до 14 тижня семестру.

3 Лабораторна робота 1: до 2 тижня семестру.

4. Лабораторна робота 2: до 3 тижня семестру.

5. Лабораторна робота 3: до 4 тижня семестру.

6. Лабораторна робота 4: до 5 тижня семестру.

7. Лабораторна робота 5: до 6 тижня семестру.

8. Лабораторна робота 6: до 8 тижня семестру.

9. Лабораторна робота 7: до 9 тижня семестру.

10. Лабораторна робота 8: до 13 тижня семестру.

Студент має право на одне перескладання кожної контрольної роботи із можливістю отримання максимально 80% початково визначених за цю контрольну роботу балів. Термін перескладання визначається викладачем.

У разі неякісного виконання лабораторної роботи, або домашнього завдання, викладач має право не зарахувати це завдання, або знизити за нього бали.

Студент має право захистити лабораторні роботи після закінчення визначеного для них терміну, але з втратою 1 балу за кожен тиждень, який пройшов з моменту закінчення терміну їх здачі.

Студент має право здавати домашні завдання роботи після закінчення визначеного для них терміну, але з втратою 0,5 балів за кожен тиждень, який пройшов з моменту закінчення терміну їх здачі.

Шкала відповідності оцінок

Відмінно / Excellent	90-100
Добре / Good	75-89
Задовільно / Satisfactory	60-74
Незадовільно / Fail	0-59

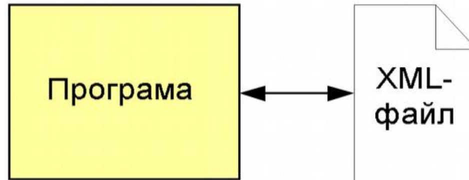
Студент допускається до складання іспиту, якщо кількість набраних за семестр балів не менше 36.

Підсумкова екзаменаційна робота – 40 балів.

Лабораторна робота № 1

Постановка завдання

Розробити програму, що забезпечує введення і редагування інформації про об'єкти відповідно до заданої предметної області. Інформація про об'єкти повинна



зберігатися в окремому файлі в форматі XML.

Вимоги до роботи

Програма не вимагає створення інтерфейсу користувача. Тестування працездатності програми здійснюється на основі сценаріїв, які демонструють можливості програми.

Відомості про всі об'єкти зберігаються в єдиному файлі XML. При запуску програми інформація про об'єкти завантажується з файлу. По завершенню роботи з об'єктами, дані записуються в файл XML. Редагування даних здійснюється в пам'яті програми.

Структура XML документа повинна перевірятися на відповідність заданому опису DTD або XMLSchema (відповідно до варіанта завдання).

Характеристики об'єктів, що автоматизуються визначаються студентом самостійно.

Обов'язковою характеристикою об'єкта є його унікальний ідентифікатор. Програма повинна забезпечувати унікальність ідентифікатора при виконанні операцій додавання і редагування об'єктів.

Наприклад, для варіанта №1, об'єкт Країна може мати характеристики:

- *код країни (унікальний ідентифікатор)*

 назва країни
а об'єкт Місто - характеристики:
 код міста (унікальний ідентифікатор)
 посилання на країну
 назва міста
 кількість жителів
 ознака столиці
Програма повинна підтримувати виконання наступних операцій з даними:
 завантаження інформації про об'єкти з файлу
 збереження інформації про об'єкти в файл
 додавання нового об'єкта
 зміна параметрів існуючого об'єкта
 видалення об'єкта
 пошук об'єктів за заданими критеріями і вивід інформації про об'єкти
Наприклад, для варіанта №1 необхідно реалізувати наступні операції:
 завантаження даних про міста і країни з файлу
 збереження даних про міста і країни в файл
 додавання нової країни
 додавання нового міста для заданої країни
 видалення міста
 видалення країни
 зміна параметрів міста і країни
 пошук міста / країни за унікальним ідентифікатором
 видача повного списку країн
 видача списку міст, що належать країні з заданим кодом
Зверніть увагу, що у всіх варіантах завдань об'єкти різних категорій знаходяться в ієрархічній
залежності. Наприклад, у варіанті №1 у кожної країни може бути кілька міст, при цьому одне
місто належить тільки одній країні.

Рекомендації по виконанню роботи

<u>Подання інформації про об'єкти в програмі</u>

Для подання інформації про об'єкти, що автоматизуються в програмі рекомендується використовувати *класи*. У наступному прикладі ми оголосимо класи на мові Python, що описують предметну область відповідно до варіанта №1.

```
class Country:
def_init_(self, id, name):
  self.id = id
  self.name = name

class City:
def_init_(self, id, name, iscap, count, country):
  self.id = id
  self.name = name
  self.iscap = iscap
  self.count = count
  self.country = country
```

Для зберігання множини об'єктів зручно використовувати масиви. Управління масивами в Python відбувається наступним чином:

```
# Оголошуємо масив країн
countries = []

# Додаємо об'єкт в масив
countries.append(Country(1,'Україна'))

# Визначаємо розмір масиву
print(len(countries))

# Звертаємося до елементу масиву
print(countries[0].name)
```

Управління об'єктами

Для управління об'єктами слід передбачити окремий клас. Даний клас буде містити масиви об'єктів і надавати методи для редагування об'єктів, а також для завантаження і збереження даних в XML-файл:

```
class WorldMap:
# Масив країн countries = []
# Масив міст cities = []

# Отримати країну з заданим кодом
def getCountry(self, id):
# повертаємо країну з заданим кодом
# якщо країни з заданим кодом в масиві countries немає –
генеруємо виняток

def getCountryInd(self, ind):
# повертаємо країну з заданим порядковим номером
# якщо номер виходить за межі індексів масива -
генеруємо виняток

# Отримати кількість країн
def countCountries(self):
# повертаємо кількість країн
# Записати дані в файл XML
def saveToFile(self, filename):
# ...

# Додати нову країну
def addCountry(self, id, name):
# якщо країни з заданим кодом в масиві countries ще немає
# додаємо нову країну в масив
# в іншому випадку генеруємо виняток

# Видалити країну
def deleteCountry(self, id):
```

```
# Видаляємо країну з заданим кодом, а також всі міста, які  посилаються на дану країну
# Якщо країни з заданим кодом в масиві countries немає – генеруємо виняток

# Додати нове місто для заданої країни
def addCity(self, id, name, isCapital, count, countryCode):
# якщо місто з заданим кодом code вже є – генеруємо виняток,  якщо країни з заданим кодом
# countryCode немає – генеруємо виняток в іншому випадку, додаємо нове місто
...
```

<u>Структура XML-файлу.</u>

Для зберігання інформації про об'єкти, що автоматизуються використовується один XML- файл. Файл має ієрархічну структуру, яка відображатиме залежності між об'єктами.

Наприклад, для варіанта №1, XML-файл може виглядати наступним чином:

```
<?Xml version = "1.0" encoding = "WINDOWS-1251"?>

<Map>
<Country id="1" name="Україна">
<City id="1" name="Київ" iscap="1" count="3000000"/>
<City id="2" name="Львів" iscap="0" count="600000"/>
</ Country>
<Country id="2" name="Білорусь">
<City id="3" name="Мінськ" iscap="1" count="1700000"/>
<City id="4" name="Вітебськ" iscap="0"
count="350000"/>
</ Country>
</Map>
```

Читання документа XML.

Обробка документа XML всередині програми Python може здійснюватися за допомогою парсерів SAX, DOM, JDOM, а також інших стандартних засобів. Далі ми подивимося, як працювати з DOM-парсером.

DOM (Document Object Model об'єктна модель документа) використовується для деревовидного представлення інформації, що зберігається в документі XML. DOM включає в себе набір інтерфейсів, що містяться в пакеті org.w3c.dom, зокрема:

- *Document* представлення документа XML
- *Element* елемент XML
- *Text* текстовий рядок

Перед роботою з документом XML, необхідно створити парсер для документу.

```
# Open XML document using minidom parser
DOMTree = xml.dom.minidom.parse("map.xml")
```

Щоб отримати кореневий елемент документа використовується:

```
collection = DOMTree.documentElement
```

Для отримання дочірніх елементів, що містяться всередині даного елемента, можна використовувати метод *getElementsByTagName()*. Даний метод повертає колекцію елементів із заданим ім'ям.

Для отримання значення атрибута використовується метод елемента *getAttribute().*

Для отримання тексту, що міститься всередині елемента (і всіх його дочірніх елементів) використовується метод *getTextContent().*

У наступному прикладі ми пройдемо по xml-документу, представленого в попередньому розділі, і

виведемо на екран інформацію про об'єкти, описані в документі.

```
# Отримуємо кореневий елемент
collection = DOMTree.documentElement

# Отримуємо колекцію країн
countries = collection.getElementsByTagName("Country")

# Проходимо по країнам
for country in countries:

# Отримуємо поточну країну
id = country.getAttribute("id") name =
country.getAttribute('name') print(id, name)

# Отримуємо колекцію міст для країни
cities = country.getElementsByTagName("City")

# Проходимо по містах
for city in cities:
# Отримуємо поточне місто
name = city.getAttribute("name") print("\t", name)
```

Результат роботи даного прикладу буде виглядати ось так:
1 Україна
Київ Львів
2 Білорусь
Мінськ Вітебськ

Валідація документа по DTD

Валідація документа XML перевірка коректності структури документа XML. Валідація може здійснюватися на основі опису DTD або схеми XML.

Для валідації документа XML за описом DTD слід налаштувати відповідним чином парсер. Для парсера слід передати обробник:

```
DOMTree = xml.dom.minidom.parse("countries.xml",
parser)
```

Перед викликом методу parse, слід створити екземпляр обробника і зареєструвати його в парсері:

```
from lxml import etree
parser = etree.XMLParser(dtd_validation=True)
```

Слід зазначити, що необхідне посилання на опис DTD повинно розміщуватися всередині документа XML:

```
<?Xml version = "1.0" encoding = "WINDOWS-1251"?>
<!DOCTYPE map SYSTEM "map.dtd">
<map>...</ map>
```

Валідація документа за схемою XML

Один з поширених способів валідації документа за схемою XML використання засобів lxml

Для валідації документа слід створити об'єкт схема класу *lxml.etree.XMLSchema* на основі необхідної схеми XML і використовувати цей об'єкт для перевірки.

```
# Створення схеми
xml_validator = lxml.etree.XMLSchema(file="schema.xsd")
```

Після виклику методу parse можна перевірити чи документ є валідний за даною схемою:

```
xml_file = lxml.etree.parse("countries.xml")
is_valid = xml_validator.validate(xml_file)
```

Формування документа XML.

Тепер подивимося, як сформувати документ XML на основі наявних даних з використанням дерева DOM.

```
# Створюємо кореневий елемент
root = minidom.Document()
```

*xml = root.createElement('**map**') root.appendChild(xml)*

 # Створюємо об'єкт "країна"
*country1 = root.createElement("**country**");*
*country1.setAttribute("**id**", "**1**");*
*country1.setAttribute("**name**", "**Україна**");*
xml.appendChild(country1)

*# Створюємо ще один об'єкт "країна" country2 = root.createElement("**country**"); country2.setAttribute("**id**", "**2**"); country2.setAttribute("**name**", "**Білорусь**");*
xml.appendChild(country2);

 # Створюємо об'єкти "міста"
*city1 = root.createElement("**city**"); city1.setAttribute("**id**", "**1**"); city1.setAttribute("**name**", "**Київ**");*
*city1.setAttribute("**count**", "**3000000**");*
*city1.setAttribute("**iscap**", "**1**"); xml.appendChild(city1);*

*city2 = root.createElement("**city**"); city2.setAttribute("**id**", "**5**"); city2.setAttribute("**name**", "**Вітебськ**");*
*city2.setAttribute("**count**", "**350000**");*
*city2.setAttribute("**iscap**", "**0**"); xml.appendChild(city2)*

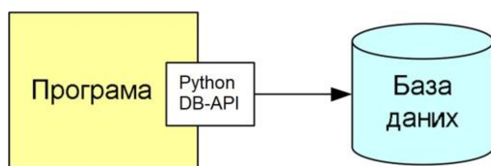
Щоб зберегти отримане дерево у файл, можна використовувати toprettyxml для XML:

xml_str = root.toprettyxml(indent="\t") save_path_file = ***"countries.xml"***
*with open(save_path_file, "**w**") as f: f.write(xml_str)*

Лабораторна робота 2

Постановка завдання

Розробити програму, що забезпечує введення і редагування інформації про об'єкти відповідно до заданої предметної області. Інформація про об'єкти повинна зберігатися в окремій базі даних. Доступ до даних здійснюється з використанням засобів DB-API.



Вимоги до роботи

Програма не вимагає створення інтерфейсу користувача. Тестування працездатності програми здійснюється на основі сценаріїв, які демонструють можливості програми.

Відомості про об'єкти зберігаються в таблицях бази даних. Читання і редагування даних здійснюється за допомогою запитів SQL.

Характеристики об'єктів, що автоматизуються визначаються студентом самостійно. Обов'язковою характеристикою об'єкта є його унікальний ідентифікатор. Унікальність ідентифікаторів при виконанні операцій додавання і редагування об'єктів повинна забезпечуватися засобами СУБД або засобами програми, що розробляється. Наприклад, для варіанта №1, об'єкт Країна може мати характеристики:

- *код країни (унікальний ідентифікатор)*

- *назва країни*

а об'єкт Місто - характеристики:

- *код міста (унікальний ідентифікатор)*
- *посилання на країну*
- *назва міста*
- *кількість жителів*
- *ознака столиці*

Програма повинна підтримувати виконання наступних операцій з даними:

- *додавання нового об'єкта*
- *зміна параметрів існуючого об'єкту*
- *видалення об'єкта*
- *пошук об'єктів по заданим критеріям і виведення інформації про об'єкти*

Наприклад, для варіанта №1 необхідно реалізувати наступні операції:

- *додавання нової країни*
- *додавання нового міста для заданої країни*
- *видалення міста*
- *видалення країни*
- *зміна параметрів міста і країни*
- *пошук міста / країни за унікальним ідентифікатором*
- *видача повного списку країн*
- *видача списку міст, що належать країні з заданим кодом*

Зверніть увагу, що у всіх варіантах завдань об'єкти різних категорій знаходяться в ієрархічній залежності. *Наприклад, у варіанті №1 у кожної країни може бути кілька міст, при цьому одне місто належить тільки одній країні.*

Рекомендований мову програмування - Python.
Рекомендований засіб доступу до даних – DB-API.
Рекомендована СУБД - MySQL v8.

Рекомендації по виконанню роботи

Встановлення та налаштовування СУБД

Для виконання лабораторної роботи потрібно встановити на комп'ютер і конфігурувати СУБД.

У даній лабораторній роботі рекомендується використовувати вільну СУБД MySQL, що розповсюджується по ліцензії GNU General Public License. Дистрибутив СУБД MySQL доступний на офіційному сайті MySQL за посиланням: *http://www.mysql.com/downloads/*.

Для встановлення MySQL запустіть інсталятор (файл виду mysql.xxxx.msi) і пройдіть всі кроки майстра встановлення, залишивши пропоновані за замовчуванням значення параметрів встановлення. По завершенню встановлення MySQL запустіть майстер конфігурації екземпляру MySQL (Instance Configuration Wizard):

У майстра конфігурації також використовуйте значення, пропоновані за замовчуванням.

Зверніть увагу на параметри зв'язку з конфігурованим екземпляром. Запам'ятайте порт TCP (3306 за замовчуванням) і використовуйте його в подальшому при підключенні до СУБД з програми.

На одному з кроків майстра потрібно ввести пароль (з підтвердженням) адміністратора екземпляра (root). Запам'ятайте введений пароль і використовуйте його в подальшому при роботі з СУБД.

Створення бази даних

У встановленої СУБД створимо нову базу даних з таблицями для зберігання об'єктів відповідно до заданої предметної област. Наприклад, для варіанта №1 буде

потрібно створити дві таблиці: таблицю країн і таблицю міст.

Адміністрування СУБД MySQL можна здійснювати через консольний додаток MySQL Command Line Client.

Для створення нової бази даних використовується команда CREATE DATABASE. У наступному прикладі ми створимо нову базу даних з назвою MAP:

CREATE DATABASE MAP;

Перед створенням таблиць необхідно підключитися до певної базі даних. Для цих цілей в MySQL використовується команда USE. Команда USE наказує MySQL використовувати зазначену базу даних за замовчуванням під час повторних запитів.

USE MAP;

Створення таблиць здійснюється за допомогою запиту CREATE TABLE, наприклад:

CREATE TABLE COUNTRIES
(ID_CO INTEGER NOT NULL, NAME CHAR (32));

CREATE TABLE CITIES (ID_CI INTEGER NOT NULL,
ID_CO INTEGER NOT NULL, NAME CHAR (32),
COUNT INTEGER, ISCAPITAL SMALLINT);

Для забезпечення унікальності ідентифікаторів, а також цілісності зв’язків між сутностями, слід використовувати первинні та зовнішні ключі:

ALTER TABLE COUNTRIES ADD PRIMARY KEY (ID_CO);

ALTER TABLE CITIES
ADD PRIMARY KEY (ID_CI);

ALTER TABLE CITIES

ADD FOREIGN KEY (ID_CO) REFERENCES COUNTRIES (ID_CO);

Робота з даними

Заповнити таблиці даними можна за допомогою SQL-оператора INSERT, наприклад:

```
INSERT INTO COUNTRIES VALUES (1, 'UKRAINE');
INSERT INTO COUNTRIES VALUES (2, 'USA');

INSERT INTO CITIES VALUES (1, 1, 'KYIV', 4612943, 1);
INSERT INTO CITIES VALUES (2, 1, 'LVIV', 543334, 0);
INSERT INTO CITIES VALUES (3, 2, 'NEW YORK', 8363710, 0);
```

Вибірка даних з таблиць здійснюється за допомогою оператора SELECT, наприклад:

```
SELECT * FROM CITIES
INNER JOIN COUNTRIES
ON CITIES.ID_CO = COUNTRIES.ID_CO;
```

Доступ до MySQL з програми Python

Для доступу до бази даних MySQL через інтерфейс DB-API потрібна бібліотека *MySQLdb* Завантажте бібліотеку прописавши в терміналі команду *pip install MySQLdb*

Підключіть до проекту бібліотеку MySQLdb, яка містить класи DB-API:

import MySQLdb

Встановлення з'єднання з базою даних з програми Python

Установка з'єднання з базою даних повинна виконуватися на початку роботи програми. Отримане

з'єднання буде використовуватися в процесі роботи програми при виконанні операцій над даними.

Управління з'єднанням здійснюється через клас Connection. Для установки з'єднання використовується метод connect. Метод приймає на вхід параметри з'єднання:

- url, який визначає протокол і ідентифікатор бази даних
- ім'я користувача (для зручності можна працювати під адміністратором: root)
- пароль
- назва бази даних

і в разі успіху повертає об'єкт класу Connection.

У разі виникнення помилки метод getConnection генерує виняток.

У наступному прикладі ми встановимо з'єднання з базою даних map, розміщеною на локальному сервері (localhost, порт 3306), використовуючи ім'я користувача root і пароль password.

```
url = "localhost"
databaseName = "map"

# Open database connection
db = MySQLdb.connect(url, "root", "password",
databaseName)
# працюю з даними
#...
# завершую з'єднання
```

Виконання запитів з програми Python

Виконання запитів SQL здійснюється через клас cursor. Об'єкт класу cursor створюється в рамках заданого з'єднання за допомогою методу cursor() класу connection:

```
cursor = db.cursor()
```

Зміна даних здійснюється через метод execute класу cursor. Метод приймає на вхід рядок із запитом SQL типу INSERT, UPDATE або DELETE і повертає кількість змінених записів в таблиці.

```
sql = "INSERT INTO COUNTRIES VALUES (3, 'CHINA')"
cursor.execute(sql)
```

Вибірка даних здійснюється за допомогою методу execute. Метод приймає на вхід SQL-запит виду SELECT. Для того щоб отримати результат слід викликати метод fetchall() , що дозволяє обробляти результуючі таблиці вибірок.

У наступному прикладі ми виберемо з таблиці COUNTRIES всі записи і виведемо їх в консоль:

```
sql = "SELECT * FROM COUNTRIES"
# Execute the SQL command
cursor.execute(sql)
# Fetch all the rows in a list of lists.
results = cursor.fetchall() for row in results:
id = row[0] name = row[1]
# Now print fetched result
print("%d\t%s" % (id, name))
```

У разі виникнення помилки методи генерують виняток.

Приклад програми Python DB-API

Наступна програма реалізує додавання, видалення і запит даних про країни, що зберігаються в таблиці COUNTRIES бази даних MAP. Кожна країна характеризується унікальним ідентифікатором і назвою.

```
import MySQLdb
```

```
url = "localhost"
databaseName = "map"

db = MySQLdb.connect(url, "root", "password",
databaseName) cursor = db.cursor()

# Запит всіх країн
def showCountries():
sql = "SELECT ID_CO, NAME FROM COUNTRIES"
try:
cursor.execute(sql)
results = cursor.fetchall() for row in results:
id = row[0] name = row[1]

print("%d\t%s" % (id, name))
except:
print("ПОМИЛКА при отриманні списку країн ")

# Додавання країни
def addCountry(id, name):
sql = "INSERT INTO COUNTRIES (ID_CO, NAME) VALUES
(%d, '%s')" %
(id, name)

try:
cursor.execute(sql) db.commit()
print("Країна %s Успішно додана!" % name) return True
except:
print("ПОМИЛКА! Країна %s не додана !" % name)
db.rollback()
return False

# Видалення країни
def deleteCountry(id):
  sql = "DELETE FROM COUNTRIES WHERE ID_CO = %d"
% id id)
```

```
try:
  cursor.execute(sql) db.commit()
  print("Країна з ідентифікатором %s успішно видалена!"
% id)
  return True
except:
  print("ПОМИЛКА при видаленні країни з
ідентифікатором %s" %)
  db.rollback()
  return False

# ТЕСТОВИЙ СЦЕНАРІЙ
showCountries()
addCountry(5, "JAPAN")
addCountry(6, "UKRAINE")
deleteCountry(3)
deleteCountry(7)
showCountries()
db.close()
```

Можливий результат роботи програми:
ПЕРЕЛІК КРАЇН:
>> 1 - USA
>> 3 - CHINA
ПОМИЛКА! Країна RUSSIA не додана!
>> Duplicate entry '1' for key 'PRIMARY' Країна JAPAN успішно додана!
Країна UKRAINE успішно додана!
Країна з ідентифікатором 3 успішно видалена! Країна з ідентифікатором 7 не знайдено!
ПЕРЕЛІК КРАЇН:
>> 1 - USA
>> 5 - JAPAN
>> 6 - UKRAINE

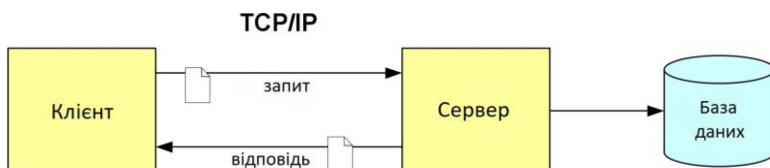
Лабораторна робота 3

Постановка задачі

На основі програми, розробленої в рамках лабораторної роботи №2, створити серверну програму, що забезпечує виконання віддалених запитів з управління об'єктами. Відомості про об'єкти повинні зберігатися в базі даних.

Розробити клієнтську програму, що відправляє серверу запити на введення, редагування і отримання інформації про об'єкти.

Взаємодія між клієнтом і сервером має здійснюватися по протоколу TCP / IP.



Вимоги до роботи

Сервер запускається у фоновому режимі і не має інтерфейсу користувача. Сервер повинен забезпечувати виконання операцій, представлених у вашому варіанті завдання.

На кожен запит клієнта сервер повинен відправляти відповідь. Відповіді сервера повинні відрізнятися в залежності від результату виконання запиту клієнта (позитивна / негативна відповідь).

Сервер орієнтований на взаємодію з єдиним клієнтом. Підтримку взаємодії сервера з декількома клієнтами

реалізовувати не потрібно.

Сервер повинен забезпечувати унікальність ідентифікаторів об'єктів при виконанніоперацій додавання і редагування.

Клієнтська програма відправляє запити серверу і демонструє відповіді користувачу. Клієнтська програма не вимагає створення інтерфейсу користувача. Тестування працездатності клієнта здійснюється на основі сценаріїв, які демонструють можливості програми.

Формат інформаційних повідомлень, якими обмінюються клієнт і сервер, визначаєтьсявідповідно до номеру варіанта

Рекомендації по виконанню роботи

Взаємодія за протоколом TCP / IP

Для взаємодії по протоколу TCP / IP в мові Python можна використовувати інтерфейс сокетів.

Термін "сокет" (socket) позначає одночасно бібліотеку мережевих інтерфейсів і термінал каналу зв'язку (точку зв'язку), через яку процес може передавати або отримувати дані.

TCP / IP-сокети дозволяють реалізувати надійні двонаправлені, орієнтовані на роботу з потоками з'єднання точка-точка між віддаленими вузлами.

В цілому алгоритм роботи системи клієнт-сервер виглядає наступним чином:

1. Сервер підключається до порту на хості і чекає з'єднання з клієнтом
2. Клієнт створює сокет і намагається з'єднатися через нього з віддаленим сервером
3. У разі встановлення з'єднання, сервер отримує сокет для взаємодії з клієнтом і переходить в режим очікування команд від клієнта

4. Клієнт формує команду і передає її через сокет сервера, після чого переходить в режим очікування відповіді

5. Сервер приймає команду через сокет, виконує її і пересилає відповідь клієнту, після чого знову переходить в режим очікування запитів від клієнта

Отже, на стороні сервера необхідно почати прослуховувати певний порт TCP / IP, для того щоб віддалений клієнт міг виконувати підключення. Це можна зробити за допомогою класу socket.

Спочатку, необхідно створити екземпляр класу socket і вказати в порт, який буде прослуховуватися. Якщо заданий порт зайнятий, буде згенеровано виняток.

```
s = socket.socket()
host = socket.gethostname() port = 12345
s.bind((host, port))
```

Далі, необхідно перевести сервер в режим очікування підключень. Робиться це за допомогою методу listen() та методу accept() класу socket. Метод listen налаштовує сервер і запускає лісенер, а метод accept виконує очікування з'єднання з боку клієнта

```
s.listen(5)
print('Waiting for a client ') c, addr = s.accept()
print ('Got connection from ', addr)
```

При необхідності можна встановити максимальний інтервал очікування клієнтських підключень. Робиться це за допомогою методу listen.

Отриманий об'єкт socket далі буде використовуватися для прийому і передачі даних клієнту. На стороні клієнта необхідно здійснити підключення до сервера. Для цього використовується клас socket. На клієнті необхідно явно

створити об'єкт класу socket і передати в конструктор установки сервера: IP-адреса сервера і порт.

```
s = socket.socket()
host = socket.gethostname() port = 12345
s.connect((host, port))
```

Створений об'єкт socket далі буде використовуватися для прийому і передачі даних сервера.

Прийом і передача даних

Обмін даними між клієнтом і сервером здійснюється через об'єкти socket. Після того, як з'єднання між клієнтом і сервером встановлено і обидві програми отримали свої об'єкти socket, вони можуть починати спілкуватися.

Щоб передати інформацію на віддалений комп'ютер програмою, необхідно здійснити запис цієї інформації в потік виводу сокета. Отримати потік виведення можна за допомогою методу getOutputStream об'єкта класу Socket.

Щоб отримати інформацію, передану з боку віддаленої програми, необхідно виконати читання з потоку введення сокета. Потік введення повертається за допомогою методу getInputStream об'єкта класу Socket.

Прийом і передача текстових даних

Для прийому/передачі текстових даних необхідно викикати методи send та recv.

У наступному прикладі ми передаємо з клієнта на сервер фразу «Hello, server».

```
s.send('Hello, server!')
```

Тепер розробимо серверний код, який отримає дане повідомлення і виведе його на екран.

```
print(c.recv(1024))
```

При читанні з потоку, сокет починає очікувати надходження даних від віддаленого партнера.

Прийом і передача двійкових даних

Для прийому / передачі двійкових даних необхідно використати метод send. У наступному прикладі ми передаємо з клієнта на сервер масив цілих чисел:

```
# Створюю масив
N = 3
A = int[N]
for i in range(N):
A[i] = 10 * i

# Передаю кількість елементів
s.send(N)

# Передаю масив

for i in range(N): s.send(A[i])
```

Тепер розробимо серверний код, який отримає дані числа і виведе їх на екран.

```
# Отримую кількість чисел
N = int(c.recv(1024))
# Отримую числа і виводжу на екран
for i in range(N):
val = c.recv(1024) print(val)
```

Формати даних

Формат інформаційних повідомлень, якими обмінюються клієнт і сервер, визначається відповідно до номером варіанта.

У завданнях пропонується використовувати такі формати повідомлень:

• двійковий;
• рядок з роздільником.

Розглянемо різні способи представлення і передачі даних на прикладі інформаційного повідомлення, що містить відомості про результати футбольного матчу. Нехай результат матчу характеризується назвами команд і кількістю голів, забитих кожною командою:

Двійковий формат

При використанні двійкового формату можливі наступні варіанти передачі інформації:

1. Множина окремих пакетів певного типу даних (даний спосіб представлений в прикладі 2.4.1). Кожен переданий пакет містить відомості про один параметр. наприклад:

```
s.send("Spain")
s.send("Italy")
s.send(str(4))
s.send(str(0))
```

2. Масив байт, що включає в себе всі поля інформаційного повідомлення

```
values = ("Spain", "Italy", 4, 0)
arr = bytearray(values)
s.sendall(arr)
```

3. Серіалізований об'єкт:

```
m = object()
m.command1 = "Spain"
m.command2 = "Italy"
m.score1 = 4
m.score2 = 0

data_string = pickle.dumps(m) s.send(data_string)
```

Рядок з роздільником

При використанні даного формату, всі поля інформаційного повідомлення записуються у текстовому вигляді в один рядок, розділені деяким службовим символом.

Наприклад, в наступному рядку як роздільник використовується символ відсотка:

"Spain% Italy% 4% 0".
values = ("Spain", "Italy", 4, 0)
packer = struct.Struct('s%s%I%I')
*packed_data = packer.pack(*values)*
s.sendall(packed_data)

Взаємодія в режимі клієнт-сервер

При взаємодії за схемою «клієнт-сервер» ініціатором взаємодії виступає клієнт. Таким чином, сервер завжди повинен знаходитися в режимі очікування, так як сервер не знає, в який момент часу від клієнта надійде черговий запит.

Клієнт відправляє серверу запит на виконання певної операції, і переходить в режим очікування відповіді. Сервер, отримує запит, виконує необхідну операцію і відправляє клієнту відповідь. Клієнт обробляє відповідь і продовжує своє нормально функціонування. Сервер ж, знову переходить в режим очікування запитів.

Приклад 1. Передача даних у двійковому форматі

Наступний простий приклад демонструє взаємодію в режимі клієнт-сервер через сокети.

Розробимо сервер, що виконує арифметичні операції над заданими числами, і клієнта, що взаємодіє з цим сервером. Сервер приймає від клієнта два числа і код арифметичної операції (будемо вважати, що 0 - це складання, 1 - віднімання), виконує над числами необхідну

дію і повертає клієнту результат. Дані, що передаються між клієнтом і сервером, представлені в двійковому форматі.

Серверний код:

```
import socket s = None
# запустити сервер
def start():
  s = socket.socket()
  c, addr = s.accept()
 # опрацювання запиту while True:
 # Отримую запит від клієнта
 oper = int(c.recv(1024)) # Операція
 v1 = int(c.recv(1024)) # Число 1
 v2 = int(c.recv(1024)) # Число 2

# Рахую результат
result = (v1 + v2) if (oper == 0) else (v1 - v2) # Відправляю
результат клієнту c.send(str(result))

try:
host = socket.gethostname() port = 12345
s.bind((host, port)) s.listen(5)
start() except:
print("Виникла помилка")
```

Клієнтський код:

```
import socket

s = socket.socket()
host = socket.gethostname() port = 12345 s.connect((host,
port))
```

```
# відправити запит серверу і отримати відповідь
def sendQuery(operation, value1, value2):
  # відправляю запит
  s.send(str(operation))
  s.send(str(value1))
  s.send(str(value2))
# отримую відповідь
  return int(s.recv(1024))
# порахувати суму чисел
def sum(value1, value2):
return sendQuery(0, value1, value2)
# порахувати різницю чисел
def sub(value1, value2):
  return sendQuery(1, value1, value2)
try:
  a = sum(15, 20) b = sub(30, 38)
  c = sum(100, 200)
  print(a)
  print(b)
  print(c)
except:
print("Виникла помилка")
```

Приклад 2. Передача даних в форматі рядку з роздільником

Модифікуємо приклад 1, щоб клієнт і сервер обмінювалися повідомленнями в форматі рядки з роздільником і ускладнити протокол взаємодії клієнта і сервера. Тепер сервер буде перевіряти, чи вірна кількість параметрів прийшла йому на вхід, чи вірний тип вхідних параметрів, і чи правильно встановлено код операції.

Протокол взаємодії буде виглядати наступним чином:

Клієнт відправляє серверу запит в форматі рядку: "Код_операції # операнд1 # операнд2"

Як роздільник використовується символ '#'.

В даному прикладі представлено два додатки клієнта. Клієнт 1 (клас Client) - модифікація клієнта з прикладу 1, що працює з текстовими повідомленнями.
Клієнт 2 (классSimpleClient) - «клієнт-шкідник» - додаток, намагається відправити серверу некоректні запити.

Вихідний код програми-сервера:

```
import socket import struct import sys

c = None
unpacker = struct.Struct('I#I#I') packer = struct.Struct('I#I')
# опрацювання запиту
def processQuery():
result = 0 # Результат операції comp_code = 0 # Код завершення try:
data = c.recv(unpacker.size) if sys.getsizeof(data) != 12:
comp_code = 1 # невірна кількість параметрів
else:
try:
unpacked_data = unpacker.unpack(data) oper = unpacked_data[0]
v1 = unpacked_data[1] v2 = unpacked_data[2]

if oper == 0:
result = v1 + v2 # операція - додавання
elif oper == 1:
result = v1 - v2 # операція - віднімання
else:
comp_code = 2 # невірний код операції
except:
comp_code = 3 # невірний тип параметрів

packed_data = packer.pack(*(comp_code, result)) c.sendall(packed_data)
return True
```

```
except :
return False


# запустити сервер
s = socket.socket()
host = socket.gethostname() port = 12345
s.bind((host, port))

while True:
# Приймаємо з'єднання від нового клієнта
s.listen(5)
c, addr = s.accept()

# Поки з'єднання активно, обробляємо запити
while processQuery(): pass
```

Клієнт 1 (вихідний код):

```
import struct
import socket
import sys

s = socket.socket()
packer = struct.Struct('I#I#I')
unpacker = struct.Struct('I#I')

def connect():
 host = socket.gethostname()
 port = 12345
 s.connect((host, port))

# відправити запит серверу і отримати відповідь
def sendQuery(operation: int, value1, value2):
```

```
packed_data = packer.pack(*(operation, value1, value2))
s.sendall(packed_data)

data = s.recv(unpacker.size)
if sys.getsizeof(data) != 41:
raise IOError("Invalid response from server")
try:
unpacked_data = unpacker.unpack(data)
comp_code = unpacked_data[0]
result = unpacked_data[1]
if comp_code == 0:
return result else:
raise IOError("Error while processing query") except IOError:
raise except:
raise IOError("Invalid response from server")

# порахувати суму чисел
def sum(value1, value2):
return sendQuery(0, value1, value2)

# порахувати різницю чисел
def sub(value1, value2):
return sendQuery(1, value1, value2)

# від'єднатися
def disconnect(): s.close()

try:
connect()

a = sum(15, 20)
b = sub(30, 38)
c = sum(10, 20)
print(a)
print(b)
print(c)
```

```
disconnect() except:
print("Виникла помилка")
```

Клієнт 2 (вихідний код):

```
import socket import struct
s = socket.socket()

packer = struct.Struct('I I I')
unpacker = struct.Struct('I I')

def connect():
host = socket.gethostname()
port = 12345
s.connect((host, port))
connect()
s.sendall(packer.pack(*("бубубу")))
print(s.recv(unpacker.size))
s.sendall(packer.pack(*('xxx', 'yyy', 'zzz')))
print(s.recv(unpacker.size))
s.sendall(packer.pack(*(10, 20, 30)))
 print(s.recv(unpacker.size))
 s.sendall(packer.pack(*(0, 5, 7)))
print(s.recv(unpacker.size))
 s.sendall(packer.pack(*(1, 5, 7)))
 print(s.recv(unpacker.size))
```

Клієнт 2 (результат роботи):

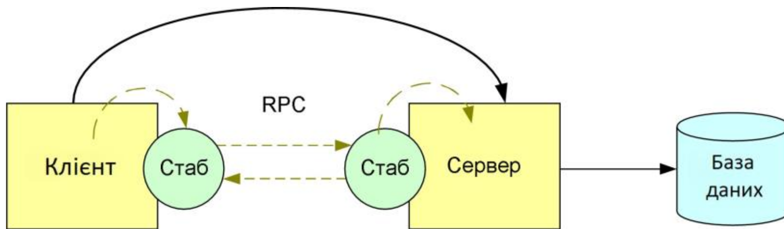
```
1, 0
3, 0
2, 0
0, 12
0, -2
```

Лабораторна робота 4

Постановка задачі

В інформаційній системі, розробленій в рамках лабораторної роботи №3, організувати взаємодію клієнта і сервера з використанням механізму віддаленого виклику процедур / методів (RPC / PYRO4).

Виклик віддаленої процедури / звернення до віддаленого об'єкту



Вимоги до роботи

Сервер запускається у фоновому режимі і не має інтерфейсу користувача. Сервер повинен підтримувати одночасну роботу з декількома клієнтами. Сервер повинен забезпечувати виконання операцій, представлених у вашому варіанті завдання.

Клієнтська програма викликає віддалені методи сервера і демонструє результати користувачеві. Клієнтська програма не вимагає створення інтерфейсу користувача. Тестування працездатності клієнта здійснюється на основі сценаріїв, які демонструють можливості ІС.

Рекомендований інтерфейс взаємодії pyro4.

Рекомендації по виконанню роботи

Технологія Pyro4

Технологія Pyro4 забезпечує віддалений виклик методів об'єктів в розподілених системах і може

використовуватися для організації синхронної клієнт-серверної взаємодії програм.

В рамках технології Pyro4 додаток-сервер створює об'єкт і реєструє його для віддаленого доступу. Клієнтські програми отримують посилання на віддалений об'єкт і працюють з ним таким чином, як ніби він є для них звичайним локальним об'єктом.

Звернення до серверного об'єкту клієнт здійснює з використанням інтерфейсу об'єкта . Інтерфейс описує серверні методи та їх параметри, але не містить реалізації методів.

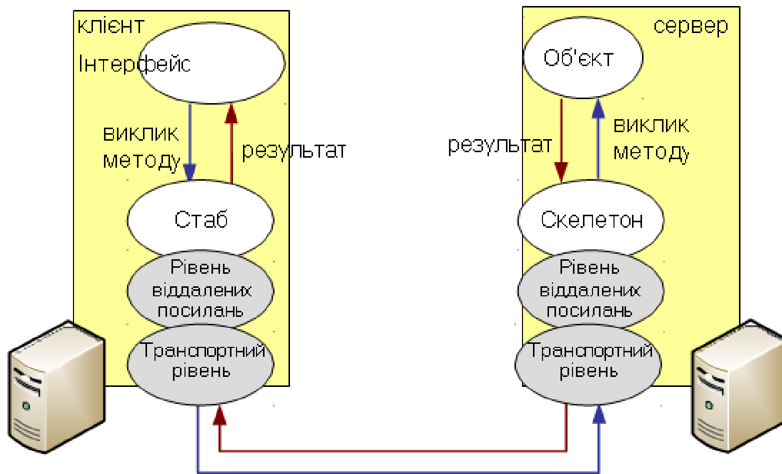
Технологія Pyro4 приховує від розробників додатків реалізацію процесу мережевої взаємодії між клієнтським і серверним додатками. За формування мережевих пакетів і їх передачу повністю відповідає проміжне ПЗ Pyro4.

Архітектура Pyro4 представлена схематично на мал.3. Клієнт, викликаючи через інтерфейс метод віддаленого об'єкта, насправді звертається до спеціалізованого локального об'єкту-заглушки стабу (stub). Стаб упаковує параметри виклику методу в байтовое повідомлення. При цьому параметри- об'єкти піддаються сереалізації. Процес пакування параметрів називається маршалінгом (marshaling). Сформоване повідомлення передається по мережі на сервер. Його отримує скелетон (skeleton) серверна заглушка. Завдання скелетона витягти параметри з отриманого повідомлення і викликати необхідний метод серверного об'єкта (який, зауважимо, є локальним для скелетона). Результат виклику методу скелетон упаковує в байтовое повідомлення і передає

назад стабу клієнта. Стаб клієнта розпаковує повідомлення і повертає результат програмі, що його викликала.

За стабом і скелетоном ховається два допоміжних рівня: рівень віддалених посилань і транспортний рівень.

Рівень віддалених посилань реалізує протокол взаємодії, незалежний від стаба і скелетона. Даний рівень знає, як потрібно керувати посиланнями клієнта на віддалені об'єкти, і яку модель передачі даних використовувати при взаємодії клієнта і сервера.

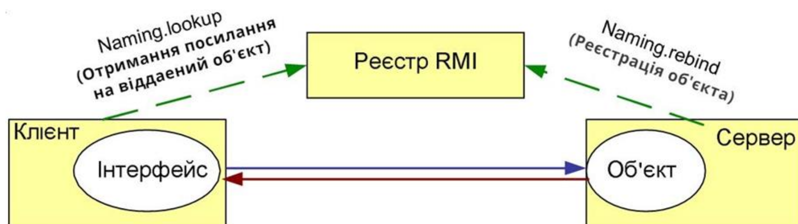


Транспортний рівень відповідає за доставку мережевих пакетів між клієнтом і сервером.

Яким чином клієнт знаходить віддалений серверний об'єкт?

З'єднання клієнта і сервера здійснюється через спеціальну службуслужбу імен та каталогів. Служба імен або каталогів зазвичай виконується на відомому всім клієнтам хості і має відомий номер порту.
Реєстр Pyro4 працює на серверній машині, яка містить об'єкти віддалених служб. Основна функція реєстру Pyro4 - надання клієнтам посилань на віддалені серверні об'єкти.

Сервер, який бажає надати свій об'єкт віддаленим програмам, попередньо реєструє його в реєстрі Pyro4 під деяким загальновідомим ім'ям. Клієнти, що бажають працювати з віддаленим об'єктом, звертаються до реєстру Pyro4 і отримують посилання на об'єкт по його імені.



Розробка клієнт-серверної системи на базі Pyro4 в найпростішому випадку передбачає виконання таких кроків:

1. Розробка класу серверного об'єкта.
2. Розробка серверного додатка (додатки, керуючого серверним об'єктом).
3. Розробка клієнтської програми (додатки, що звертається до віддаленого серверного об'єкту).

При роботі клієнт-серверної системи повинен дотримуватися наступний порядок дій:

1. Запускається служба реєстру Pyro4 (це можна зробити як в самому серверному додатку, так і в окремому процесі).
2. Створюється серверний об'єкт і реєструється в реєстрі Pyro4.
3. Запускаються клієнтські програми, отримують з реєстру Pyro4 посилання на віддалений об'єкт, і викликають методи об'єкта.

<u>Розробка класу серверного об'єкта</u>

У наступному прикладі ми реалізуємо клас

@Pyro4.expose class PyroServer:
def hello(self):
return "Hello from server!"

Метод hello класу PyroServer повертає клієнту рядок з привітанням від сервера: "Hello from server!".

Розробка серверного додатка

Серверна програма реєструє об'єкт в сервісі і отримує його адресу для підключення. За цим адресом клієнт зможе до нього підключитись

Нижче представлений приклад вихідного серверного коду, що створює свій локальний реєстр:

daemon = Pyro4.Daemon()

uri = daemon.register(PyroServer) ns = Pyro4.locateNS()
ns.register('obj', uri)

daemon.requestLoop()

!!! Перед виконанням програми-сервера, що використовує зовнішній реєстр, необхідно запустити службу реєстру Pyro4. Зробити це можна з командного рядка, виконавши програму Pyro-ns,

Зупинити службу реєстру Pyro4 можна натиснувши клавіші CTRL + C.

Розробка клієнтської програми

Клієнтська програма звертається до віддаленого реєстру Pyro4 і отримує посилання на серверний об'єкт. Для роботи з об'єктом (для виклику віддалених методів) клієнтський додаток використовує інтерфейс серверного об'єкта.

Нижче представлений приклад клієнта, що викликає метод hello віддаленого серверного об'єкта.

```
ns = Pyro4.locateNS()
uri = ns.lookup('obj')
o = Pyro4.Proxy(uri)
print(o.hello())
```

Зверніть увагу на назву об’єкта, який передається у метод lookup

Приклади

Приклад 1: Калькулятор

Система складається з клієнтського і серверного додатків. Сервер управляє об'єктом, що надає клієнтам віддалені методи, що обчислюють суму і різницю чисел.

Додаток-сервер: Клас серверного об'єкта Calculator

```
import Pyro4
@Pyro4.expose
class Calculator:
def sum(self, x, y):
 return x + y

def sub(self, x, y):
 return x – y
```

Додаток-сервер: Головний клас

```
import Pyro4

from calculator
import Calculator
daemon = Pyro4.Daemon()
uri = daemon.register(Calculator)
 ns = Pyro4.locateNS()
ns.register('calculator', uri)
```

daemon.requestLoop()

Додаток-клієнт:

import Pyro4

ns = Pyro4.locateNS()
uri = ns.lookup('calculator')
o = Pyro4.Proxy(uri)
print(o.sum(10, 20))
print(o.sub(10, 4))

Приклад 2: Калькулятор (з передачею об'єкта в якості параметра)
Сервер надає клієнтам віддалені методи, що обчислюють суму і різницю чисел. Операнди для обчислень передаються через об'єкт класу Operands.

Клас Operands

class Operands:
def init (self, x, y): self.x = x
self.y = y

Додаток-сервер: Клас серверного об'єкта CalculatorImpl
import Pyro4

@Pyro4.expose
class Calculator:
def sum(self, o):
return o.x + o.y

def sub(self, o):
return o.x - o.y

Додаток-сервер: Головний клас

```
import Pyro4

from Calculator
import Calculator
daemon = Pyro4.Daemon()
uri = daemon.register(Calculator)
ns = Pyro4.locateNS()
ns.register('calculator', uri)

daemon.requestLoop()
```

Додаток-клієнт:

```
import Pyro4

from operands import Operands
ns = Pyro4.locateNS()
uri = ns.lookup('calculator')
 o = Pyro4.Proxy(uri)
operand = Operands(20, 10)

print(o.sum(operand))
print(o.sub(operand))
```

Приклад 3: Віддалений масив

На сервер зберігається масив цілих чисел.
Сервер надає клієнтам віддалені методи, що дозволяють додавати числа в масив і обчислювати суму елементів масиву.
Необхідно запобігти одночасний доступ декількох потоків до серверного масиву (така ситуація можлива в разі, якщо декілька клієнтів одночасно будуть викликати методи сервера). Для цього використовується синхронізація.
В даному прикладі реалізовано два клієнта. Клієнт 1 в нескінченному циклі запитує у сервера суму елементів масиву і виводить її на екран. Клієнт 2 в нескінченному

циклі запитує у сервера додавання в масив нового елемента.

Додаток-сервер: Клас серверного об'єкта MyArray

```
from threading
import Lock
import Pyro4

@Pyro4.expose
class MyArray:
elements = []
 lock = Lock()

def add(self, other):
 with self.lock:
self.elements.append(other)

def sum(self):
with self.lock:
sum = 0
for x in self.elements:
 sum += x
return sum
```

Додаток-сервер: Головний клас

```
import Pyro4

from Calculator
import Calculator
 daemon = Pyro4.Daemon()
uri = daemon.register(Calculator)
ns = Pyro4.locateNS()
 ns.register('array', uri)
daemon.requestLoop()
```

Додаток-клієнт 1: Головний клас

```
import Pyro4
```

```
ns = Pyro4.locateNS()
uri = ns.lookup('array')
o = Pyro4.Proxy(uri)
while True:
print(o.sum())
```

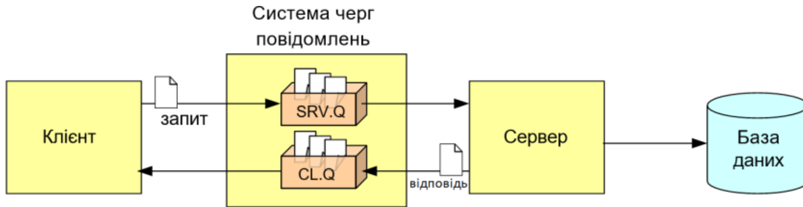
Додаток-клієнт 2: Головний клас

```
import Pyro4
ns = Pyro4.locateNS()
uri = ns.lookup('array')
o = Pyro4.Proxy(uri)
while True:
o.add(1)
```

Лабораторна робота 5

Постановка задачі

В інформаційній системі, розробленій в рамках лабораторної роботи №3, організувати асинхронну взаємодію клієнта і сервера через черги повідомлень.



Вимоги до роботи

Взаємодія клієнта і сервера здійснюється за схемою «точка-точка» (підтримка одночасних з'єднань з декількома клієнтами не вимагається) в асинхронному режимі:

- запускається клієнт, відправляє серверу кілька запитів і завершує свою роботу;
- запускається сервер, обробляє запити клієнта, відправляє клієнту відповіді і завершує свою роботу;
- знову запускається клієнт і отримує відповіді від сервера.

Взаємодія між клієнтом і сервером здійснюється через систему черг повідомлень. В системі черг повідомлень потрібно створити дві черги:

- SRV.Q
- CL.Q

серверна черга, призначена для зберігання запитів, що надійшли від клієнта клієнтська черга, призначена для зберігання відповідей на запити клієнта

Сервер запускається у фоновому режимі і не має інтерфейсу користувача. Сервер повинен забезпечувати

виконання операцій, представлених у вашому варіанті завдання.

На кожен запит клієнта сервер повинен відправляти відповідь. Відповіді сервера повинні відрізнятися в залежності від результату виконання запиту клієнта (позитивна / негативна відповідь).

Клієнтська програма відправляє запити серверу і демонструє відповіді користувачу. Клієнтська програма не вимагає створення інтерфейсу користувача. Тестування працездатності клієнта здійснюється на основі сценаріїв, які демонструють можливості програми.

Формат інформаційних повідомлень, якими обмінюються клієнт і сервер, визначається відповідно до номеру варіанта.

Введення в системи черг повідомлень

Основні поняття

Message Oriented Middleware (MOM) «Проміжне програмне забезпечення, орієнтоване на повідомлення» загальноприйнята назва програмних засобів, що забезпечують взаємодію додатків в розподіленій системі на основі механізму обміну повідомленнями. Поняття MOM також широко застосовується для позначення технології обміну даними в формі повідомлень.

Повідомлення (англ. message) структура даних, яка використовується для представлення інформації, переданої між додатками. Повідомлення складається з заголовка і прикладної частини. У заголовку міститься службова інформація про повідомлення. Заголовок повідомлення зазвичай має фіксований розмір і строго визначену структуру. Прикладна частина може містити будь-яку інформацію. Розмір, структура і формат прикладної частини визначається додатком, що створює повідомлення.

Черга (англ. queue) структура даних, призначена для проміжного зберігання інформаційних повідомлень,

переданих між додатками. Черга організовує повідомлення в порядку

FIFO (first in first out, повідомлення зберігаються в черзі в порядку свого надходження), або в порядку пріоритетів повідомлень. Прикладні програми можуть записувати повідомлення в чергу, і зчитувати повідомлення з черги. Можна сказати, що черга виступає в якості «поштової скриньки» для інформації, що передається між додатками.

Система черг повідомлень (англ. message queuing system) програмне забезпечення категорії MOM, яке керує чергами повідомлень. Система черг повідомлень забезпечує доступ до черги з боку прикладних програм, керує операціями читання і запису повідомлень, і відповідає за фізичне зберігання даних, що містяться в черзі. На сьогоднішній день на світовому ринку програмного забезпечення існує великий вибір систем черг повідомлень: WebSphere MQ компанії IBM, MSMQ компанії Microsoft, Advanced Queuing компанії Oracle, FioranoMQ компанії Fiorano Software, Sonic MQ компанії Progress Software, Active MQ спільноти Apache та інші.

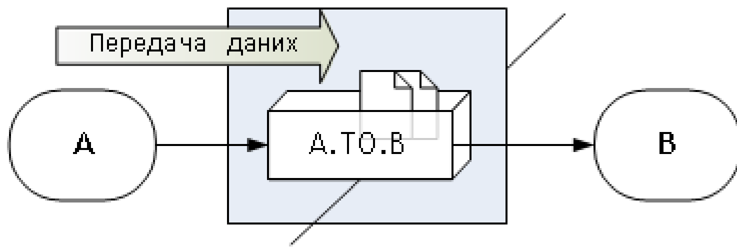
Моделі взаємодії додатків

У розподілених інформаційних системах, побудованих на базі засобів MOM, додатки можуть обмінюватися інформацією різними способами.

Нижче розглядається декілька типових моделей взаємодії додатків з використанням черг повідомлень.

Одностороння передача даних

Один з найпростіших способів взаємодії це одностороння передача даних між двома додатками: A і B.



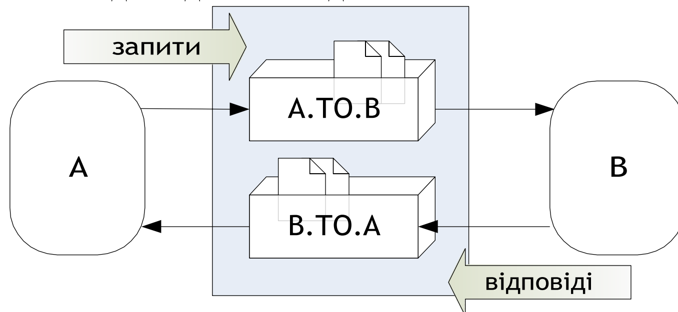
Додаток А поміщає повідомлення, що містять деяку інформацію в чергу, яку обслуговує додаток В. Додаток В послідовно витягує ці повідомлення з черги, і обробляє їх.

Дану модель взаємодії іноді називають моделлю «відправив-забув». Додаток А, відправляючи повідомлення, ніби забуває про нього. Воно не чекає відповідного повідомлення і не вимагає підтвердження доставки або обробки повідомлення.

<u>Модель «запит-відповідь»</u>

Модель «запит-відповідь» передбачає двосторонній обмін інформацією між додатками.

Додаток, відправляючи повідомлення, очікує отримання відповідного повідомлення.



Показано взаємодію додатків А і В за принципом «запит- відповідь». Для кожної з програм в системі черг повідомлень створена черга, яку вони будуть обслуговувати.

Додаток А, бажаючи відправити деякий запит додатку В, поміщає відповідне повідомлення в чергу програми В (A.TO.B). Додаток В читає дане повідомлення, обробляє запит, що міститься в повідомленні, і поміщає відповідь у чергу програми А (B.TO.A). Додаток А зчитує цю відповідь і дізнається результат обробки свого запиту.

Синхронна і асинхронна взаємодія

Взаємодія за принципом «запит-відповідь» може здійснюватися в синхронному або асинхронному режимі.

Синхронна взаємодія, як випливає з назви, має на увазі послідовний обмін повідомленнями між додатками: один додаток відправляє іншому додатку запит і чекає відповіді на нього; у тому випадку якщо протягом якогось заданого часу відповіді не отримано, додаток-відправник завершується помилкою.

Синхронна взаємодія має ряд обмежень. По-перше, синхронний обмін даними можливий лише за умови одночасної активності взаємодіючих додатків. Програми, що виконуються по різному тимчасовим регламентом, не можуть спілкуватися в синхронному режимі. По-друге, синхронній взаємодії додатків може перешкоджати відсутність постійної або якісної лінії зв'язку. По-третє, додаток, який бере участь в синхронному обміні інформацією, після відправлення запиту блокується на час очікування відповіді і стає недоступним для інших програм.

Асинхронна взаємодія не вимагає блокування додатків при очікуванні відповідей. Даний тип взаємодії здійснюється в наступний спосіб. Додаток А відправляє свій запит додатку В і не чекає негайної відповіді від нього, а продовжує своє звичайне функціонування. Після цього, додаток А буде періодично перевіряти, чи не отримав він відповідь. Паралельно, додаток А може відправити ще один запит програмі В, або брати участь у взаємодії з іншими програмами. Додаток В в той момент,

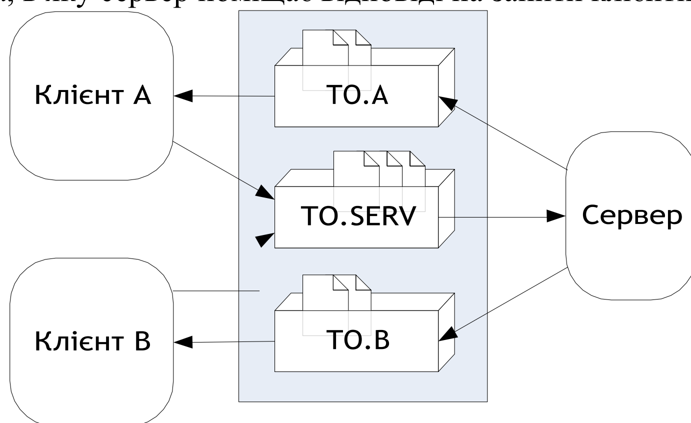
коли вважатиме за потрібне, обробить запит від програми А і відправить йому відповідь.

В асинхронній взаємодії можуть брати участь програми, які виконуються в різний час. Крім того, даний спосіб взаємодії підходить і для обміну даними при наявності неякісної лінії зв'язку

Модель «клієнт-сервер»

Модель «запит-відповідь» є окремим випадком загальної моделі «клієнт-сервер». Модель «клієнт-сервер» передбачає наявність централізованої програми сервера, що надає певний набір послуг, а також множині програм клієнтів, які звертаються до ці послуги.

У сервера є централізована черга, в яку поміщають свої запити клієнти. У кожного клієнта є своя персональна черга, в яку сервер поміщає відповіді на запити клієнтів.



Показано взаємодію трьох програм: клієнтів А і В і сервера С. Клієнти відправляють свої повідомлення в чергу сервера TO.SERV. Сервер обробляє запити клієнтів, і повертає персональну відповідь клієнту в його чергу (TO.A або TO.B).

Клієнт-серверна взаємодія може здійснюватися як в синхронному, так і в асинхронному режимі.

Модель «публікація-підписка»

«Публікація-підписка» технологія асинхронної взаємодії, що використовується для доставки повідомлень всередині групи додатків.

Додатки, обмінюються інформацією за технологією «публікація-підписка », поділяються на дві категорії:

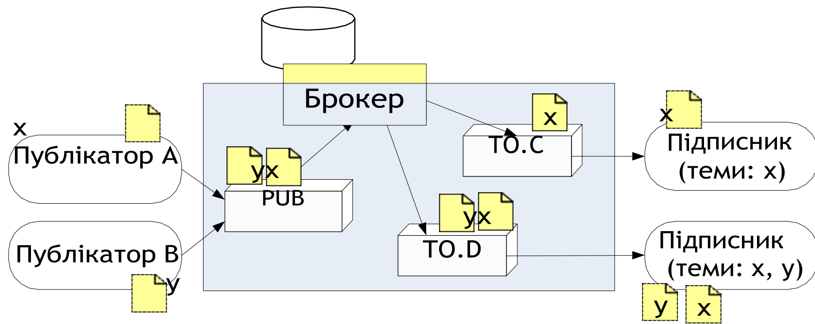
• публікатори (англ. publishers) додатки, які поширюють інформацію;

• підписники (англ. subscribers) додатки, які зацікавлені в отриманні інформації.

Взаємодією публікаторів та підписників керує спеціальна програма брокер повідомлень (англ. message broker). Розподіл повідомлень між підписниками здійснюється на основі тем повідомлень. Тема повідомлення (англ. topic) спеціальний ідентифікатор, що дозволяє віднести повідомлення до тієї чи іншої логічної категорії

Додатки-підписники реєструються всередині брокера повідомлень, вказуючи при цьому, які теми повідомлень їх цікавлять. Публікатори відправляють брокеру інформаційні повідомлення, із зазначенням теми цих повідомлень. Брокер для кожного отриманого повідомлення визначає підписників і висилає їм копію цього повідомлення.

На малюнку показаний приклад взаємодії додатків за схемою «публікація-підписка». Є два додатки-публікатора (A і B), і два додатка-підписника (C і D). При цьому додаток C зацікавлений у повідомленнях на тему x, а додаток D за темами x і y. Відомості про підписників зберігаються в базі даних брокера повідомлень.



Додаток А публікує в системі нове повідомлення по темі x, поміщаючи його в чергу брокера PUB. Аналогічним чином, додаток В публікує повідомлення по темі у. Брокер отримує два нових повідомлення, і записує копії цих повідомлень в черзі відповідних підписників (TO.C і TO.D).

<u>Підготовка Python-проекту для pymqi</u>
Завантажте бібліотеку прописавши в терміналі команду

pip install pymqi
До програмних модулів підключіть бібліотеку pymqi:

import pymqi

Установка з'єднання з менеджером черг З'єднанням з менеджером черг керує клас pymqi.
Для встановлення з'єднання з менеджером черг слід створити об'єкт класу QueueManager.
При цьому в конструктор класу передається назва менеджера черг.

QueueManager(queueManagerName)

Приклад:

qmgr = pymqi.QueueManager('QM1')
... # робота з менеджером черг

Також можна передати параметри для менеджера черг, використовуючи метод connect
qmgr = pymqi.connect(queue_manager, channel, conn_info)

Обробка помилок
Для обробки помилок, що виникають при роботі з WebSphere MQ, використовується клас pymqi.MQMIError. Клас MQMIError використовується в конструкції try catch.
Клас має два основних поля:

- comp
- reason

код завершення операції код помилки

Поле completionCode може приймати значення:

- MQCC_WARNING

операція виконалася з попередженнями

- MQCC_FAILED

операція завершилася помилкою

Поле reason використовується для уточнення причини виникнення помилки.

try:
#операції з об'єктами WebSphere MQ

```
except pymqi.MQMIError as e:
if e.comp == pymqi.CMQC.MQCC_FAILED and e.reason ==
pymqi.CMQC.MQRC_NO_MSG_AVAILABLE:

pass else:
# Some other error condition.
raise
```

Відкриття черги

Щоб відправляти і читати повідомлення, необхідно попередньо відкрити потрібну чергу. Для відкриття черги використовується метод Queue екземпляра класу pymqi По завершенню роботи з чергою її слід закрити за допомогою методу close.

У наступному прикладі ми встановимо з'єднання з менеджером черг QM1 і відкриємо чергу MY.QUEUE для читання і запису повідомлень:

```
# Установка з'єднання з менеджером QM1
MQQueueManager
qmgr = pymqi.QueueManager('QM1')
# Відкриття черги
putq = pymqi.Queue(qmgr, 'MY QUEUE')
# робота з чергою
...
# Завершення роботи
putq.close()
```

Робота з повідомленням

Для роботи з повідомленнями WebSphere MQ може використовуватися будь-який клас і будь яка форма передачі даних. Для цього можна використати клас, який створювався в лабораторній роботі номер 4. Також можна пересилати кожне значення окремо, як це буде здійснено в прикладі

Запис повідомлення в чергу

Для запису повідомлення в чергу використовується метод put класу Queue. У метод передається повідомлення, яке необхідно помістити в чергу.

В наступному прикладі ми запишемо тестове повідомлення в чергу MY.QUEUE:

qmgr = pymqi.QueueManager('QM1') putq = pymqi.Queue(qmgr, 'MY QUEUE') putq.put('Hello from Python!')

Читання повідомлення з черги

Щоб прочитати повідомлення з черги використовується метод get класу Queue. Метод витягує повідомлення з черги і записує його в заданий об'єкт.

В наступному прикладі ми прочитаємо тестове повідомлення із черги MY.QUEUE:

qmgr = pymqi.QueueManager('QM1') putq = pymqi.Queue(qmgr, 'MY QUEUE') print(putq.get())

При читанні повідомлення з черги можна вказати додаткові опції, використовуючи клас GMO Даний клас дозволяє:

- визначити інтервал очікування повідомлення в черзі;
- встановити використання транзакцій при читанні повідомлення;
- визначити критерії пошуку повідомлення в черзі;
- встановити режим перегляду повідомлень (повідомлення будуть прочитані без видалення з черги).

В наступному прикладі ми встановимо інтервал очікування надходження повідомлення в чергу:
gmo = pymqi.GMO()

```
gmo.Options = pymqi.CMQC.MQGMO_WAIT |
pymqi.CMQC.MQGMO_FAIL_IF_QUIESCING
gmo.WaitInterval = 5000 # 5 seconds
putq.get(gmo)
```

Приклад взаємодії через черги

Розробимо клієнт-серверну систему «Калькулятор». Клієнт передає серверу запити на виконання арифметичних операцій над двома числами

Сервер виконує обчислення і повертає клієнту відповідь. Взаємодія здійснюється в асинхронному режимі через черги повідомлень за схемою «запит-відповідь» (сервер працює з єдиним клієнтом).

Взаємодія клієнта і сервера відбувається за наступним сценарієм:

1. Додаток-клієнт запускається вперше (метод Client.main1) і відправляє серверу кілька запитів на виконання арифметичних обчислень, після чого завершує свою роботу.
2. Запускається сервер. Він отримує запити клієнта, проводить необхідні обчислення, формує і відправляє відповіді, і завершує свою роботу.
3. Клієнт запускається знову (метод Client.main2) і отримує результати обчислень від сервера.

Вихідний код програми-сервера

```
import pymqi

# Налаштовую інтервал очікування 3 сек
gmo = pymqi.GMO()
gmo.Options = pymqi.CMQC.MQGMO_WAIT |
pymqi.CMQC.MQGMO_FAIL_IF_QUIESCING
gmo.WaitInterval = 3000

qmgr = pymqi.QueueManager('QM1')
```

```
 putq = pymqi.Queue(qmgr, 'SRV.Q')
getq = pymqi.Queue(qmgr, 'CL.Q')

# Обробка повідомлення-запиту

def processQuery(): try:
# Читаю повідомлення з черги запитів
oper = getq.get(gmo)
# Операція
v1 = getq.get(gmo)
# Число1
v2 = getq.get(gmo)
# Число2
# Рахую результат
response = v1 + v2
if (oper == 0)
 else v1 - v2
# Формую відповідь
putq.put(oper)
putq.put(v1)
putq.put(v2)
 putq.put(response)
return True
except:
return False


# Обробляю всі запити
i = 0
while (processQuery()):
i += 1

print("Опрацьовано" + i + "Запитів")
 putq.close()
getq.close()
```

Вихідний код програми-клієнта

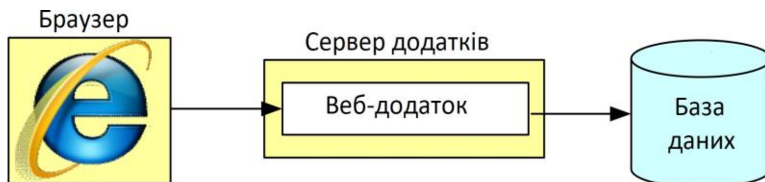
```
import pymqi

# Налаштовую інтервал очікування 3 сек
gmo = pymqi.GMO()
gmo.Options = pymqi.CMQC.MQGMO_WAIT |
pymqi.CMQC.MQGMO_FAIL_IF_QUIESCING
gmo.WaitInterval = 3000

qmgr = pymqi.QueueManager('QM1') getq =
pymqi.Queue(qmgr, 'SRV.Q') putq = pymqi.Queue(qmgr,
'CL.Q')


# відправити запит серверу
def sendQuery(operation, value1, value2): putq.put(operation)
putq.put(value1) putq.put(value2)


# порахувати суму чисел
def sum(value1, value2):
 sendQuery(0, value1, value2)


# порахувати різницю чисел
def sub(value1, value2):
sendQuery(1, value1, value2)


# Отримати відповідь від сервера
def printResult():
try:
# Читаю повідомлення з черги запитів
oper = getq.get(gmo)
```

```
# Операція
v1 = getq.get(gmo)
 # Число1

v2 = getq.get(gmo)
 # Число2
response = getq.get(gmo)
s = '+'
if (oper == 0)
else '-' print(v1 + s + v2 + '=' + response)
return True
except:
return False


def main1():
 try:
sum(15, 20)
sub(30, 38)
sum(100, 200)
except:
pass


def main2():
while (printResult()): pass


main1() # або main2
```

Лабораторна робота 6

Постановка задачі

Розробіть веб-додаток, що надає користувачеві відомості про об'єкти (відповідно до заданої предметної області). Інформація про об'єкти зчитується з бази даних, створеної в рамках лабораторної роботи №2.



Вимоги до роботи

Веб-додаток виконується на сервері додатків. Доступ до додатка здійснюється через стандартний браузер.

Додаток динамічно формує веб-сторінку на основі даних, що зберігаються в базі. Зовнішній вигляд веб-сторінки повинен задаватися за допомогою таблиць стилів в окремому файлі CSS.

Рекомендована СУБД — MySQL v8.

Рекомендований засіб доступу до даних — Pydb.

Рекомендоване середовище розробки програми — Intellij PyCharm.

Рекомендований фреймворк — Django.

Рекомендації по виконанню роботи

Встановлення фреймворка

Для виконання лабораторної роботи потрібно встановити на комп'ютер фреймворк. У даній роботі рекомендується використати безкоштовний фреймоврк django. Для встановлення необхідно прописати в терміналі:

pip install django

Створення веб-додатка в середовищі PyCharm

Розглянемо процес створення веб-додатка на прикладі середовища PyCharm:

1. Запустіть середовище PyCharm і викличте з головного меню команду створення нового проекту. Створіть пустий проект.
2. В терміналі пропишіть:

django-admin startproject MyProject

Ця команда створить django проект з усіма необхідними бібліотеками і скриптами для запуску. Веб-додаток створено.

Створений в середовищі PyCharm веб-додаток django за замовчуванням включає в себе стартову сторінку.

3. Відкрийте веб-додаток, ввівши в терміналі команду:

python manage.py runserver

PyCharm відкомпілює проект, і встановить веб додаток на сервер додатків. Перевірити працездатність додатку можна через будь-який стандартний браузер, вказавши в адресному рядку URI додатка. наприклад:
http://127.0.0.1:8000/

де:

- 127.0.0.1 мережеве ім'я сервера
- 8000 — порт сервера додатків

Розробка додатку

В django кожна категорія сайту представляє з себе окремий додаток. Додаток в свою чергу це модуль в якому є набір файлів, і кожен файл відповідає за певні дії, наприклад один відповідає за відображення сторінки, другий за стилі, третій за перехід по урл і тд.

Створення додатку в PyCharm

Для створення нового додатку в середовищі PyCharm введіть в терміналі команду:

python manage.py startapp main

У проекті буде створено нова папка main. Новий додаток слід зареєструвати в settings.py. Для цього слід в INSTALLED_APPS дописати щойно створений додаток.

```
INSTALLED_APPS = [
'django.contrib.admin', 'django.contrib.auth',
'django.contrib.contenttypes', 'django.contrib.sessions',
'django.contrib.messages', 'django.contrib.staticfiles', 'main'
]
```

Відслідковування URL

Щоб відслідковувати перехід користувача на певну сторінку, потрібно додати цю сторінку в файл urls.py. Додамо перехід на головну сторінку. Для цього створимо перехід в основному проекті на щойно створений проект написавши в urls.py наступне:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
path('admin/', admin.site.urls), path('', include('main.urls'))
]
```

В main папці слід створити файл urls.py і в ньому написати наступне:

```
from django.urls import path
from myproj.main import views urlpatterns = [
path('', views.index)
]
```

Виведення html на екран

Ми прописали виклик методу views.index, коли користувач відкриє головну сторінку. Тепер потрібно прописати цей метод. Він приймає на вхід request і виводить html код. Ось код файлу views.py

```
from django.http import HttpResponse
```

```
def index(request):
return HttpResponse("<h4>Hello , World!!!<h4>")
```

Приклад простої сторінки

Створимо метод, що формує веб-сторінку з таблицею множення. Для оформлення таблиці будемо використовувати стилі CSS.

Метод (файл views.py)

```
from django.http import HttpResponse

def multiplicationTable(request): text = """
<html>
<head>
<title> Таблиця множення </title>
<link href = \"/newcss.css\" rel = \"stylesheet" type =
\"text/css\">
</head>
<body>
<h1> Таблиця множення </h1>
"""

# виводжу ТАБЛИЦЮ
text += "<table>"

# Виводжу заголовок
text += "<tr>"
text += "<th> </th>"
for i in range(2, 10):
text += "<th>" + str(i) + "</th>"
text += "</tr>"

# Виводжу вміст
for i in range(2, 9): text += "<tr>"
text += "<th>" + str(i) + "</th>"
```

```
for j in range(2, 10):
text += "<td>" + str(i * j) + "</td>"
text += "</tr>"

text += "</table>" text += "</body>" text += "</html>"
return HttpResponse(text)
```

Таблиця стилів (файл newcss.css)

```
table {
font-size: 14pt;
border: 1px solid black;
}

th {
width: 50px;
height: 20px;
text-align: center;
background-color: greenyellow; border: 1px solid black;
}

td {
width: 50px;
height: 20px;
text-align: center; border: 1px solid black;
```

розширений варіант на сайті Django Web Framework (Python) - Learn web development | MDN (mozilla.org)

Лабораторна робота 7

Постановка задачі

Розробіть веб-сервіс на базі SOAP, що надає користувачеві відомості про об'єкти (відповідно до заданої предметної області). Інформація про об'єкти зчитується з бази даних, створеної в рамках лабораторної роботи №2.

SOAP (англ. Simple Object Access Protocol) — протокол обміну структурованими повідомленнями в розподілених обчислювальних системах, базується на форматі XML.

Спочатку SOAP призначався, в основному, для реалізації віддаленого виклику процедур (RPC), а назва була абревіатурою: Simple Object Access Protocol — простий протокол доступу до об'єктів. Зараз протокол використовується для обміну повідомленнями в форматі XML, а не тільки для виклику процедур. SOAP є розширенням мови XML-RPC.

SOAP можна використовувати з будь-яким протоколом прикладного рівня: SMTP, FTP, HTTP та інші. Проте його взаємодія з кожним із цих протоколів має свої особливості, які потрібно відзначити окремо. Найчастіше SOAP використовується разом з HTTP.

SOAP є одним зі стандартів, на яких ґрунтується технологія веб-сервісів.

SOAP веб-сервіс — це базова однонаправлена модель з’єднання, що забезпечує узгоджену передачу повідомлення від відправника до одержувача, потенційно допускає наявність посередників, які можуть обробляти частину повідомлення або додавати до нього додаткові елементи.

SOAP є найголовнішою частиною технології Web-сервісів, здійснює перенесення даних по мережі з одного місця в інше.

SOAP забезпечує доставку даних веб-сервісів. SOAP дозволяє відправнику і одержувачу XML-документів

підтримувати загальний протокол передачі даних, що забезпечує ефективність мережевого зв'язку.

Як працює? Протокол SOAP не розрізняє виклик процедури і відповідь на неї, а просто визначає формат послання (message) у вигляді документу XML. Послання може містити виклик процедури, відповідь на неї, запит на виконання якихось інших дій або просто текст. Специфікацію SOAP не цікавить вміст послання, вона задає тільки його оформлення.

1. Запит, який протокол SOAP надсилає, надходить на сервер, до якого можна отримати доступ за допомогою об'єкта сервера.
2. Інтерфейс користувача створює деякі файли або методи, що складаються з об'єкта сервера та імені інтерфейсу для об'єкта сервера. Вона також містить іншу інформацію, таку як ім'я інтерфейсу та методи.
3. Протокол SOAP використовує HTTP для відправки XML на сервер за допомогою методу POST, сервер аналізує запрос і надсилає результат клієнту.
4. Сервер створює більше XML, що складаються з відповідей на запит інтерфейсу користувача за допомогою HTTP.

Приклади використання.

Використовуйте SOAP API для створення, отримання, оновлення або видалення записів, таких як облікові записи, похідні та визначені користувачем об'єкти. Ви також можете застосовувати SOAP API для керування паролями, виконання пошуків тощо, використовуючи SOAP API на будь-якій мові, яка підтримується веб-службами.

Які засоби надані SOAP?

- PutAddress(): використовується для введення адреси на веб-сторінці і має примірник адреси в запиті SOAP.

- PutListing(): використовується для вставки повного XML-документа на веб-сторінку. Він отримує XML-файл як аргумент і переносить XML-файл у зв'язок XML-парсера, який читає його і вставляє його у запит SOAP як параметр.
- GetAddress(): використовується для отримання імені запиту і отримує результат, який найкраще відповідає запиту. Ім'я надсилається до запиту SOAP у формі символьного рядка тексту.
- GetAllListing(): використовується для повернення повного списку у формат XML.

Які існують підходи розробки SOAP веб-сервісів?

Стосовно розробки SOAP веб-служб, розрізняють два різних способи описаних нижче:

- Підхід першого контракту («Contract-first approach»): контракт спочатку визначається XML і WSDL, а потім використовуються Java-класи для реалізації цього контакту.
- Підхід останнього контакту («Contract-last approach»): Спочатку визначаються Java-класи, а потім створюється контракт, який зазвичай є WSDL-файлом, що генерується з класу Java.

Найпопулярнішим підходом є метод «Contract-first».

Приклад створення SOAP веб-сервісу на мові програмування Python

Тепер розглянемо приклад створення SOAP веб-сервісу на мові програмування Python. Для цього нам знадобляться наступні речі :

- Django 1.8
- Spyne 2.10
- Suds 0.4

Почнемо зі створення файлу views.py, в якому і буде реалізований наш веб- сервіс. Для прикладу ми маємо функцію hello, яка виводить на екран значення

«Hello, {input name}», та функцію sum, яка повертає результат додавання двох чисел. Код з даного файлу:

```
# views.py
from django.views.decorators.csrf
import csrf_exempt from spyne.application
import Application
from spyne.decorator
import rpc
from spyne.model.primitive
import Unicode, Integer
from spyne.protocol.soap import Soap11
from spyne.server.django import DjangoApplication
from spyne.service import ServiceBase

class SoapService(ServiceBase):
@rpc(Unicode(nillable=False), _returns=Unicode)
def hello(ctx, name):
return 'Hello, {}'.format(name)


@rpc(Integer(nillable=False), Integer(nillable=False),
_returns=Integer)
def sum(ctx, a, b):
return int(a + b)

soap_app = Application( [SoapService],
tns='django.soap.example',
in_protocol=Soap11(validator='lxml'),
out_protocol=Soap11(),
)

django_soap_application = DjangoApplication(soap_app)
my_soap_application =
csrf_exempt(django_soap_application)
```

rpc - це декоратор методів для позначення методу як віддаленого виклику процедури в підкласі *spyne.service.ServiceBase.*

@rpc (Unicode (nillable = False), _returns = Unicode) над *def hello (ctx,name):* означає, що є один вхідний параметр який вважається рядком Unicode, не може бути нульовим; вихід також є рядком Unicode.

@rpc (Integer (nillable = False),
Integer (nillable = False), _returns = Integer) над *sum def (ctx, a, b):* означає, що є 2 параметри введення, і обидва вони не мають нульових цілих чисел, вихід також є цілим числом.

Додаток SOAP повинен бути створений для обгортки SOAP сервісів, сервіси повинні бути вказані як перший аргумент spyne.application.Application конструктора, основою нашої реалізації список є SOAP сервісом [SoapService]. Якщо у вас є кілька сервісів, аргументом має бути [SoapService, AnotherSoapService]. tns це
- targetNameSpace.

Створимо також файл urls.py

```
# urls.py
from django.conf.urls import patterns, url import views
urlpatterns = patterns( '', url(r'^soap_service/',
views.my_soap_application),
)
```

Для цієї реалізації WSDL URL має бути наступним: http://127.0.0.1:8000/soap_service/?WSDL або http://127.0.0.1:8000/soap_service/?wsd l, тобто це означає, що WSDL не чутливий до регістру.

Для демонстрації функціоналу створимо також SOAP клієнт.

```
# soap_client.py # coding=utf-8

from suds.client import Client from suds.cache import NoCache

my_client = Client('http://127.0.0.1:8000/soap_service/?WSDL', cache=NoCache())
print my_client
print 'Function hello: ', my_client.service.hello('Minh')
print 'Function sum: ', my_client.service.sum(10, 20)
```

Ми створили простий SOAP клієнт та сервер. Результат запуску команди

```
python soap_client.py:
Suds ( https://fedorahosted.org/suds/ ) version: 0.4 GA build: R699-20100913

Service ( SoapView ) tns="django.soap.example" Prefixes (1)
ns0 = "django.soap.example" Ports (1):
(Application)
Methods (2): hello(xs:string name, )
sum(xs:integer a, xs:integer b, ) Types (4):
hello helloResponse sum sumResponse
Function hello: Hello, Minh Function sum: 30
```

Також ви можете побачити дані WSDL файлу:


```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions
xmlns:plink="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
```

```
xmlns:s12enc="http://www.w3.org/2003/05/soap-encoding/"
xmlns:s12env="http://www.w3.org/2003/05/soap-envelope/"
xmlns:senc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:senv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="django.soap.example"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressi
ng" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xop="http://www.w3.org/2004/08/xop/include"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
targetNamespace="django.soap.example"
name="Application">
 <wsdl:types>
<xs:schema targetNamespace="django.soap.example"
elementFormDefault="qualified">
<xs:complexType name="hello">
<xs:sequence>
<xs:element name="name" type="xs:string"
minOccurs="0"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="helloResponse">
<xs:sequence>
<xs:element name="helloResult" type="xs:string"
minOccurs="0" nillable="true"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="sumResponse">
<xs:sequence>
<xs:element name="sumResult" type="xs:integer"
minOccurs="0" nillable="true"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="sum">
<xs:sequence>
```

```
<xs:element name="a" type="xs:integer"
minOccurs="0"/>
<xs:element name="b" type="xs:integer"
</xs:sequence>
</xs:complexType>
<xs:element name="helloResponse"
type="tns:helloResponse"/>
<xs:element name="sum" type="tns:sum"/>
<xs:element name="hello" type="tns:hello"/>
<xs:element name="sumResponse"
type="tns:sumResponse"/>
</xs:schema>
</wsdl:types>
<wsdl:message name="sum">
<wsdl:part name="sum" element="tns:sum"/>
</wsdl:message>
<wsdl:message name="sumResponse">
<wsdl:part name="sumResponse"
element="tns:sumResponse"/>
</wsdl:message>
<wsdl:message name="hello">
<wsdl:part name="hello" element="tns:hello"/>
</wsdl:message>
<wsdl:message name="helloResponse">
<wsdl:part name="helloResponse"
element="tns:helloResponse"/>
</wsdl:message>
<wsdl:service name="SoapService">
<wsdl:port name="Application" binding="tns:Application">
<soap:address
location="http://ewalletdev.airpay.vn:8001/soap_service/"/>
</wsdl:port>
</wsdl:service>
<wsdl:portType name="Application">
<wsdl:operation name="sum" parameterOrder="sum">
```

```
<wsdl:input name="sum" message="tns:sum"/>
<wsdl:output name="sumResponse"
message="tns:sumResponse"/>
</wsdl:operation>
<wsdl:operation name="hello" parameterOrder="hello">
<wsdl:input name="hello" message="tns:hello"/>
<wsdl:output name="helloResponse"
message="tns:helloResponse"/>
</wsdl:operation>
</wsdl:portType>
<wsdl:binding name="Application" type="tns:Application">
<soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="sum">
<soap:operation soapAction="sum" style="document"/>
<wsdl:input name="sum">
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output name="sumResponse">
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="hello">
<soap:operation soapAction="hello" style="document"/>
<wsdl:input name="hello">
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output name="helloResponse">
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
</wsdl:definitions>
```

Лабораторна робота 8

Постановка задачі

Розробіть веб-сервіс на базі REST, що надає користувачеві відомості про об'єкти (відповідно до заданої предметної області). Інформація про об'єкти зчитується з бази даних, створеної в рамках лабораторної роботи №2.

RESTful Web Service це Web Service написаний на основі структури REST. REST вже широко використовується і замінює Web Service, які грунтуються на SOAP і WSDL. RESTful Web Service легкий (lightweigh), його легко розширити і підтримувати.

Перші поняття про REST (REpresentational State Transfer) були введені в 2000 році в докторській дисертації Roy Thomas Fielding (співзасновник HTTP). У дисертації він детально знайомить нас з обмеженнями, правилами, способами виконання в системі для отримання системи REST.

REST визначає правила архітектури для дизайну ваших Web services, фокусується на систематичних ресурсах, включаючи якого формату стан ресурсів і передається по HTTP, і написаний різними мовами. Якщо порахувати за кількістю веб сервісів, які використовуються, REST став популярним за минулі роки як сервіс моделі дизайну з великою перевагою. Насправді, REST має великий вплив і майже замінив SOAP і WSDL так як його набагато простіше і легше використовувати.

REST це набір правил для створення Web Service, який слідує 4 основним правилам дизайну:

- Використовувати явні методи HTTP
- Не має стану
- Показує структуру папок як URls
- Передача JavaScript Object Notation (JSON), XML або обох.

У слові RESTful, ful є суфіксом, в англійській мові схоже слово help означає допомогу, то слово helpful це корисний.

Приклад створення REST веб-сервісу на мові програмування Python

У Python існує багато фреймворків для веб-розробки, але Django (вимовляється як yango) та Flask виділяється з натовпу. В даному прикладі буде використовуватись перший.

Ми створимо простий API, щоб дозволити адміністраторам переглядати та редагувати користувачів та групи в системі.

Налаштування проекту

Створіть новий проект Django під назвою tutorial, а потім запустіть новий додаток під назвою quickstart.

Макет проекту повинен виглядати так:

```
$ pwd
<some path>/tutorial
$ find .
/manage.py
./tutorial
./tutorial/ init .py
./tutorial/quickstart
./tutorial/quickstart/ init .py
./tutorial/quickstart/admin.py
./tutorial/quickstart/apps.py
./tutorial/quickstart/migrations
./tutorial/quickstart/migrations/ init .py
./tutorial/quickstart/models.py
./tutorial/quickstart/tests.py
./tutorial/quickstart/views.py
./tutorial/settings.py
./tutorial/urls.py
./tutorial/wsgi.py
```

Це може виглядати незвично, що додаток створено в каталозі проектів. Використання простору імен проекту дозволяє уникнути сутичок імен із зовнішніми модулями. Тепер синхронізуйте свою базу даних вперше:
python manage.py migrate

Ми також створимо початкового користувача з іменем admin з паролем password123. Пізніше в нашому прикладі ми підтвердимо автентифікацію цього користувача.
python manage.py createsuperuser --email admin@example.com --username admin

Серіалізатори

Спочатку ми будемо визначати деякі серіалізатори. Створимо новий модуль під назвою tutorial / quickstart / serializers.py, який ми будемо використовувати для представлення даних.

```
from django.contrib.auth.models import User, Group from
rest_framework import serializers
class UserSerializer(serializers.HyperlinkedModelSerializer):
class Meta:
model = User

fields = ['url', 'username', 'email', 'groups']

class
GroupSerializer(serializers.HyperlinkedModelSerializer):
class Meta:
model = Group

fields = ['url', 'name']
```

Зверніть увагу, що в цьому випадку ми використовуємо гіперпосилання з HyperlinkedModelSerializer. Ви також можете використовувати первинний ключ та різні інші

відносини, але гіперпосилання - це хороший RESTful дизайн.

Представлення

Відкрийте tutorial/quickstart/views.py і введіть наступний код.

```
from django.contrib.auth.models import User, Group
from rest_framework import viewsets
from rest_framework import permissions
from tutorial.quickstart.serializers import UserSerializer,
GroupSerializer
class UserViewSet(viewsets.ModelViewSet):
"""
API endpoint that allows users to be viewed or edited.
"""
queryset = User.objects.all().order_by('-date_joined')
serializer_class = UserSerializer
permission_classes = [permissions.IsAuthenticated]
class GroupViewSet(viewsets.ModelViewSet):
"""
API endpoint that allows groups to be viewed or edited.
"""
queryset = Group.objects.all()
serializer_class = GroupSerializer
permission_classes = [permissions.IsAuthenticated]
```

Замість того, щоб писати декілька представлень, ми групуємо всю загальну поведінку в класи, що називаються ViewSets. Ми можемо легко розділити їх на окремі представлення, якщо нам потрібно, але використання наборів представлень підтримує логіку подання добре організованою, а також дуже стислою.

URLs

Далі підключимо URLs API. У файлі tutorial/urls.py...

```
from django.urls import include, path
from rest_framework import routers
from tutorial.quickstart import views router =
routers.DefaultRouter()
router.register(r'users', views.UserViewSet)
router.register(r'groups', views.GroupViewSet)
# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
path('', include(router.urls)),
path('api-auth/', include('rest_framework.urls',
namespace='rest_framework'))

]
```

Оскільки ми використовуємо набори viewsets замість представлень, ми можемо автоматично генерувати URL-conf для нашого API, просто
зареєструвавши viewsets переглядів у класі маршрутизатора. Знову ж таки,
якщо нам потрібен більший контроль над URL-адресами API, ми можемо просто перейти до використання звичайних представлень на основі класів та явного написання URL-conf.

Pagination

Pagination дозволяє контролювати кількість повернених об'єктів на сторінку. Щоб увімкнути його, додайте наступні рядки до tutorial / settings.py

```
REST_FRAMEWORK = {
'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination',
'PAGE_SIZE': 10
}
```

Налаштування
Додайте 'rest_framework' до INSTALLED_APPS. Модуль налаштувань буде знаходитись у tutorial/settings.py

```
INSTALLED_APPS = [
...
'rest_framework',

]
```

Тестування нашого API

Тепер ми готові протестувати створений нами API. Давайте запустимо сервер із командного рядка.

```
python manage.py runserver
```

Тепер ми можемо отримати доступ до нашого API, як з командного рядка, використовуючи такі інструменти, як curl ...

```
bash: curl -H 'Accept: application/json; indent=4' -u
admin:password123 http://127.0.0. 1:8000/users/

{
"count": 2,
"next": null, "previous": null, "results": [
{
"email": "admin@example.com", "groups": [],
"url": "http://127.0.0.1:8000/users/1/",
"username": "admin"
},
{
"email": "tom@example.com", "groups": [   ],
"url": "http://127.0.0.1:8000/users/2/", "username": "tom"
}
]
}
```

Або за допомогою інструмента командного рядка http ...

bash: http -a admin:password123 http://127.0.0.1:8000/users/

HTTP/1.1 200 OK
{

"count": 2,
"next": null, "previous": null, "results": [
{
"email": "admin@example.com", "groups": [],
"url": "http://localhost:8000/users/1/",
"username": "paul"
},
{
"email": "tom@example.com", "groups": [ ],
"url": "http://127.0.0.1:8000/users/2/", "username": "tom"
}
]
}
Або безпосередньо через браузер, перейшовши за URL-адресою http:
//127.0.0.1: 8000 / users / ...

ПЕРЕЛІК ВЖИВАНИХ СКОРОЧЕНЬ

CLR Common Language Runtime – вихідний файл виконання мов;

JSON JavaScript Object Notation – Об'єктна нотація JavaScript.

HTML HyperText Markup Language – мова розмітки гіпертекстових документів;

IDE Integrated development environment – інтегроване середовище розробки;

XML Extensible Markup Language – розширювана мова розмітки;

БД база даних;

ЖЦ життєвий цикл;

ООП об’єктно-орієнтоване програмування;

ПС програмна система.

РЕКОМЕНДОВАНІ ДЖЕРЕЛА

Основні:

1. M. Lutz: Learning Python: Powerful Object-Oriented Programming, 5th ed. // O'Reilly Media, Inc., 2013.
2. D. Beazley, B.K. Jones: Python Cookbook: Recipes for Mastering Python 3, 3rd ed. // O'Reilly Media, Inc., 2013.
3. A. Martelli, A. Ravenscro, S. Holden: Python in a Nutshell: The Definitive Reference, 3rd ed. // O'Reilly Media, Inc., 2017.
4. B. Lubanovic: Introducing Python: Modern Computing in Simple Packages. // O'Reilly Media, Inc., 2015.
5. J. Hunt: A Beginners Guide to Python 3 Programming. // Springer, 2019.
6. J. Hunt: Advanced Guide to Python 3 Programming. // Springer, 2019.
7. N. Ceder: The Quick Python Book, 3rd ed. // Manning Publications Co., 2018.
8. D. Hellmann: The Python 3 Standard Library by Example, 2nd ed. // Pearson Education, Inc., 2017.
9. C. Hattingh: Using Asyncio in Python 3: Understanding Python's Asynchronous Programming Features. // O'Reilly Media, Inc., 2018.
10. F. Romano, D. Phillips, R. van Hattem: Python: Journey from Novice to Expert. // Packt Publishing, 2016.
11. M. Jaworski, T. Ziade: Expert Python Programming: Become an ace Python programmer by
learning best coding practices and advance-level concepts with Python 3.5, 2nd ed. // Packt Publishing, 2016.
12. H.J.W. Percival: Test-Driven Development with Python: Obey the Testing Goat: Using Django, Selenium and JavaScript. // O'Reilly Media, Inc., 2014.
13. J. Kronika, A. Bendoraitis: Django 2 Web Development Cookbook: 100 practical recipes on
building scalable Python web apps with Django 2, 3rd ed. // Packt Publishing, 2018.

14. B. Okken: Python Testing with pytest: Simple, Rapid, Effective, and Scalable. // The Pragmatic Programmers, LLC, 2017.
15. G.L. Turnquist, B.N. Das: Python Testing Cookbook: Easy solutions to test your Python projects using test-driven development and Selenium, 2nd ed. // Packt Publishing, 2018.
16. B.M. Harwani: Qt5 Python GUI Programming Cookbook: Building responsive and powerful cross-platform applications with PyQt. // Packt Publishing, 2018.

Додаткові:

1. L. Ramalho: Fluent Python: Clear, Concise, and Effective Programming. // O'Reilly Media, Inc., 2015.
2. K. Reitz, T. Schlusser: The Hitchhiker's Guide to Python: Best Practices for Development. // O'Reilly Media, Inc., 2016.
3. B. Stephenson: The Python Workbook: A Brief Introduction with Exercises and Solutions, 2nd ed. // Springer, 2019.
4. A.B. Downey: Think Python: How to Think Like a Computer Scientist, 2nd ed. // O'Reilly Media, Inc., 2016.
5. D. Phillips: Python 3 Object-oriented Programming: Unleash the power of Python 3 objects, 2nd ed. // Packt Publishing, 2015.
6. I. Kalb: Learn to Program with Python 3: A Step-by-Step Guide to Programming, 2nd ed. // Apress, 2018.
7. J.B. Browning, M. Alchin: Pro Python 3: Features and Tools for Professional Development, 3rd ed. // Apress, 2019.
8. M.L. Hetland: Beginning Python: From Novice to Professional, 3rd ed. // Apress, 2017.
9. P. Gries, J. Campbell, J. Montojo: Practical Programming, Third Edition: An Introduction to Computer Science Using Python 3.6. // The Pragmatic Bookshelf, 2017.
10. M. Gorelick, I. Ozsvald: High Performance Python: Practical Performant Programming for Humans. // O'Reilly Media, Inc., 2014.
11. M. Lutz: Python Pocket Reference: Python in Your Pocket, 5th ed. // O'Reilly Media, Inc., 2014.

12. M. Summerfield: Python in Practice: Create Better Programs Using Concurrency, Libraries, and Patterns. // Addison-Wesley, 2014.
13. A. Mele: Django 2 by Example: Build powerful and reliable Python web applications from scratch. // Packt Publishing, 2018.
14. T. Aratyn: Building Django 2.0 Web Applications: Create enterprise-grade, scalable Python web applications easily with Django 2.0. // Packt Publishing, 2018.
15. F. Marani: Practical Django 2 and Channels 2: Building Projects and Applications with Real-Time Capabilities. // Apress, 2019.
16. J. Elman, M. Lavin: Lightweight Django: Using Rest, Websockets & Backbone. // O'Reilly Media, Inc., 2015.
17. A. Pajankar: Python Unit Test Automation: Practical Techniques for Python Developers and Testers. // Apress, 2017.
18. B. Oliveira: Pytest Quick Start Guide: Write better Python code with simple and maintainable tests. // Packt Publishing, 2018.
19. S. Govindaraj: Test-Driven Python Development: Develop high-quality and maintainable Python applications using the principles of test-driven development. // Packt Publishing, 2015.

Електронні ресурси.

1. The Official Home of the Python Programming Language (https://www.python.org/)
2. Python 3/8 Documentation (https://docs.python.org/3.8/)
3. PEP 0 – Index of Python Enhancement Proposals (https://www.python.org/dev/peps/)
4. Repl.it – Online Python Editor and IDE (https://repl.it/languages/python3)
5. PyQt (https://riverbankcomputing.com/software/pyqt/intro)
6. PyTest Framework (https://docs.pytest.org/)
7. Django Web framework (https://www.djangoproject.com/)
8. PostgreSQL (https://www.postgresql.org/

Навчальне видання

М.М. Верес, Л.О. Катеринич, О.О. Супрун

Програмування мовою Python

Навчальний посібник