**Лабораторна робота №7**

# Тема. Об'єктно-орієнтоване програмування на Python 3. Конструктор класса – метод \_\_init\_\_()

**Мета роботи**. Об'єктно-орієнтоване програмування на Python 3.

**Зміст**.

# Вивчення відомостей про об'єктно-орієнтоване програмування на Python 3. Конструктор класса – метод \_\_init\_\_().

1. Виконання роботи.
2. Отримання результату.

**Ключові положення.**

Цикли, розгалуження і функції — все це елементи процедурного програмування. Його можливостей цілком достатньо для написання невеликих, простих програм. Однак великі проекти часто реалізують, використовуючи парадигму об'єктно-орієнтованого програмування (ООП).

Припустимо, команда програмістів займається розробкою гри. Програму-гру можна представити як систему, яка складається з цифрових героїв і середовища їх існування, яке включає багато предметів. Кожен воїн, зброя, дерево, будинок — це цифровий об'єкт, в якому "упаковано" його властивості і дії, за допомогою яких він може змінювати свої властивості і властивості інших об'єктів.

Кожен програміст може розробляти свою групу об'єктів. Розробникам достатньо домовитись між собою тільки про те, як об'єкти будуть взаємодіяти між собою.

Ключову різницю між програмою, написаною в структурному стилі, і об'єктно-орієнтованою програмою можна висловити так: у першому випадку на перший план виходить логіка, розуміння послідовності виконання дій для досягнення поставленої цілі. У другому — важливіше представити програму як систему об'єктів, які взаємодіють.

Основними поняттями в ООП є клас, об'єкт, успадкування, інкапсуляція і поліморфізм.

**Поняття –клас**. Тип int — це клас цілих чисел. Числа 5, 100134, -10 і т. д. — це конкретні об'єкти цього класу. В Python об'єкти також прийнято називати екземплярами.

**Поняття — успадкування**. Нехай є клас столів, який описує загальні властивості усіх столів. Однак можна розділити усі столи на письмові, обіді і журнальні і для кожної групи створити свій клас, який буде спадкоємцем загального класу, але також вносити ряд своїх особливостей. Таким чином, загальний клас буде батьківським, а класи груп — дочірніми.

Дочірні класи успадковують особливості батьківських, однак доповнюють або у деякій мірі модифікують їх характеристики. Коли ми створюємо конкретний екземпляр стола, то повинні обрати, до якого класу столів він буде належати. Якщо він належить класу журнальних столів, то отримає усі характеристики загального класу столів і класу журнальних столів. Але не особливості письмових і обідніх.

**Поняття—Інкапсуляція.** Приховування даних, тобто неможливість напряму отримати доступ до внутрішньої структури об'єкта, так як це небезпечно. Інший смисл інкапсуляції — об'єднання властивостей і поведінки в одне ціле, тобто в клас.

**Поняття—** **Поліморфізм.** Об'єкти різних класів, з різною внутрішньою реалізацією можуть мати однакові інтерфейси. Наприклад, для чисел є операція додавання, яка позначається знаком +. Однак ми можемо визначити клас, об'єкти котрого також будуть підтримувати операцію, яка позначається цим знаком. Але це зовсім не означає, що об'єкти повинні бути числами, і буде отримуватись якась сума. Операція + для об'єктів нашого класу може означати щось інше. Але інтерфейс, в даному випадку це знак +, у чисел і нашего класу буде однаковим. Поліморфність же проявляється у внутрішній реалізації і результаті операції.

### Створення класів і об'єктів[¶](http://ruslan.rv.ua/python-essential/oop/create/#_1)

В мові програмування Python класи створюють за допомогою команди **class**, після якої вказують ім'я класу, потім ставиться двокрапка, далі з нового рядка і з відступом реалізується тіло класу:

class Ім’я класу:

pass

Якщо клас є дочірнім, то батьківські класи перераховуються у круглих дужках після імені класу:

class MyClass(ParentClass):

pass

Об'єкт створюється шляхом виклику класу за його іменем. При цьому після імені класу обов'язково ставляться дужки. Оскільки у програмному коді важливо не згубити посилання на щойно створений об'єкт, то зазвичай його пов'язують зі змінною. Отже створення двох об'єктів найчастіше виглядає так:

>>> **class** A:

... **pass**

...

>>> a = A()

>>> b = A()

### Клас як модуль

В Python клас можна представити подібно модулю. Так само як у модулі у ньому можуть бути свої змінні зі значеннями і функції. Так само як у модулі у класу є власний простір імен, доступ до якого можливий через ім'я клас:

>>> class В:

... n = 5

... def adder(v):

... return v + В.n

...

>>> В.n

5

>>> В.adder(4)

9

>>>

Однак у випадку класів використовується дещо інша термінологія. Імена, визначені в класі, називаються атрибутами цього класу. В вищенаведеному прикладі імена n і adder — це атрибути класу B. Атрибути-змінні часто називають полями і деколи властивостями. Атрибути-функції називаються методами. Кількість полів і методів у класі може бути довільною.

### Клас як створювач об'єктів[¶](http://ruslan.rv.ua/python-essential/oop/create/#_3)

>>> 1 = В()

>>> 1.n

5

>>> 1.adder(4)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: add() takes 1 positional argument but 2 were given

>>>

Інтерпретатор повідомляє нам, що adder() приймає тільки один аргумент, а було передано два. Звідки ж взявся другий аргумент, і хто він такий, якщо у дужках було вказано тільки одне число 4?

Клас створює об'єкти, які у певному сенсі є його спадкоємцями. Це означає, що якщо у об'єкта немає власного поля n, то інтерпретатор шукає його на один рівень вище, тобто у класі. Таким чином, якщо ми присвоюємо об'єкту поле с таким самим ім'ям як у класі, то воно перекриває, або перевизначає, поле клас:

>>> 1.n = 10

>>> 1.n

10

>>> В.n

5

>>>

Тут змінні 1.n і В.n — це різні змінні. Перша знаходиться у просторі імен об'єкта 1, друга — у просторі класу B. Якщо б ми не додали поле n до об'єкта a, то інтерпретатор піднявся б вище по дереву наслідування і прийшов би у клас, де би і знайшов це поле.

Щодо методів, то вони також наслідуються об'єктами клас. У даному випадку у об'єкта a немає свого власного метода add, отже, він шукається в клас Adder. Однак від клас Adder може бути породжено багато об'єктів. Методи ж найчастіше призначаються для обробки об'єктів. Таким чином, коли викликається метод, у нього треба передати конкретний об'єкт, який він буде обробляти.

Зрозуміло, що екземпляр, що передається — це об'єкт, до якого застосовується метод. Вираз 1.adder(100) виконується інтерпретатором наступним чином:

1. Шукаю атрибут adder() у об'єкта 1. Не знайшов.
2. Тоді йду шукати у клас B, так як він створив об'єкт 1.
3. Тут знайшов метод. Передаю йому об'єкт, до якого цей метод треба застосувати, і аргумент, що вказано у дужках.

Іншими словами, вираз

1.adder(100)

перетвоюється у вираз

B.adder(1, 100)

Таким чином, інтерпретатор спробував передати у метод adder() класу B два аргумента — об'єкт 1 і число 100. Але ми запрограмували метод adder() так, що він приймає тільки один параметр. В Python визначення методів не передбачають прийняття об'єкта як зрозуміле за замовчуванням. Об'єкт що приймається треба вказувати явно.

За згодою у Python для посилання на об'єкт використовується ім'я self. Ось так повинен виглядати метод adder(), якщо ми плануємо викликати його через об'єкти:

>>> class B:

... n = 5

... def adder(self, v):

... return v + self.n

...

>>>

Змінна self зв'язується з об'єктом, до якого було застосовано даний метод, і через цю змінну ми отримуємо доступ до атрибутів об'єкта. Коли цей же метод застосовується до іншого об'єкта, то self зв'яжеться вже з саме цим іншим об'єктом, і через цю змінну будуть вилучатись тільки його поля.

Давайте протестуємо оновлений метод:

>>> l = B()

>>> m = B()

>>> l.n = 10

>>> l.adder(3)

13

>>> m.adder(4)

9

Тут від класу B створюється два об'єкта – 1 та m. Для об'єкта 1 заводиться власне поле n. Об'єкт m, не має такого поля, отже успадковує n від класу B. Переконаємось у цьому:

>>> m.n **is** B.n

True

>>> l.n **is** B.n

False

У методі adder() вираз self.n – це звернення до поля n, переданого об'єкта, і не важливо, на якому рівні наслідування його буде знайдено.

### Зміна полів об'єкта[¶](http://ruslan.rv.ua/python-essential/oop/create/#_4)

Прийнято привласнювати поля, а також отримувати їх значення, шляхом виклику методів:

>>> **class** User:

... **def** setName(self, n):

... self.name = n

... **def** getName(self):

... **try**:

... **return** self.name

... **except**:

... **print**("No name")

...

>>> first = User()

>>> second = User()

>>> first.setName("Bob")

>>> first.getName()

'Bob'

>>> second.getName()

No name

Методи називають сетерами (set – встановить) та гетерами (get – отримати).

# Конструктор класу – метод \_\_init\_\_()

В ООП конструктором класу називають метод, який автоматично викликається при створенні об'єктів. Його також можна назвати конструктором об'єктів класу. Ім'я такого метода зазвичай регламентується синтаксисом конкретної мови програмування. В Python роль конструктора виконує метод \_\_init\_\_(). Об'єкт створюється в момент виклику класу по імені, і в цей момент викликається метод **init**(), якщо його визначено в класі.

Необхідність конструкторів пов'язана з тим, що часто об'єкти повинні мати власні властивості одразу. Припустимо маємо клас Person, об'єкти котрого обов'язково повинні мати ім'я та прізвище. Якщо клас буде описано наступним способом:

**class** Person:

**def** setName(self, n, s):

self.name = n

self.surname = s

то створення об'єкта можливе без полів. Для встановлення імені метод set\_name() необхідно викликати окремо:

>>> **from** test **import** Person

>>> p1 = Person()

>>> p1.setName("Bill", "Ross")

>>> p1.name, p1.surname

('Bill', 'Ross')

Наявність конструктора не дозволить створити об'єкт без полів:

**class** Person:

**def** \_\_init\_\_(self, n, s):

self.name = n

self.surname = s

p1 = Person("Sam", "Baker")

**print**(p1.name, p1.surname)

Тут при виклику класу в круглих дужках передаються значення, котрі будуть присвоєні параметрам метода **init**(). Перший параметр – self – посилання на сам щойно створений об'єкт.

Буває, що необхідно допустити створення об'єкта, якщо деякі дані в конструктор не передаються. У такому випадку параметрам конструктора класу задаємо значення за замовчуванням:

**class** Rectangle:

**def** \_\_init\_\_(self, w = 0.5, h = 1):

self.width = w

self.height = h

**def** square(self):

**return** self.width \* self.height

rec1 = Rectangle(5, 2)

rec2 = Rectangle()

rec3 = Rectangle(3)

rec4 = Rectangle(h = 4)

**print**(rec1.square())

**print**(rec2.square())

**print**(rec3.square())

**print**(rec4.square())

### Деструктор класа[¶](http://ruslan.rv.ua/python-essential/oop/constructor/#_2)

Окрім конструктора об'єктів в ООП є зворотній йому метод – деструктор. Він викликається, коли об'єкт знищується.

В Python об'єкт знищується, коли зникають усі пов'язані з ним змінні або їм присвоюється інше значення, в результаті чого зв'язок з старим об'єктом втрачається. Видалити змінну також можна за допомогою del.

В Python функцію деструктора виконує метод \_\_del\_\_(). Але в Python деструктор використовується рідко, інтерпретатор і без нього добре впорається зі "сміттям".

**Лабораторне завдання**

1. Є клас Person, конструктор якого приймає три параметри (не враховуючи self) - ім'я, прізвище і оцінку студента. Оцінка має значення задане за замовчуванням, рівне одиниці.
2. У класу Person є метод, який повертає рядок, що включає в себе всю інформацію про студента.
3. Клас Person містить деструктор, який виводить на екран фразу "Ви отримали стипендію ..." (замість три крапки повинні виводитися ім'я та прізвище об'єкта).
4. В основній гілці програми створіть три об'єкти класу Person. Подивіться інформацію про студента і об’явить, що він отримав стипендію.
5. В кінці програми додайте функцію input (), щоб скрипт не завершився сам, поки не буде натиснуто Enter. Інакше ви відразу побачите як видаляються всі об'єкти при завершенні роботи програми.