МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНИЙ ПОСІБНИК

ПРОГРАМУВАННЯ МОВОЮ PYTHON

(основи, інклюзивний курс)

Київ 2022

УДК 681.3.07

Гриф надано Навчально-науковим інститутом інформаційних технологій Державного університету телекомунікацій (протокол №3 від 27.10. 2021 року.)

Рецензенти: Складанний П.М., кандидат технічних наук, доцент, завідувач кафедри інформаційної та кібернетичної безпеки імені Володимира Бурячка факультету інформаційних технологій та управління Київського університету імені Бориса Грінченка

Трінтіна Н.А., Негоденко О.В., Гаманюк І.М., Шевченко С.М.

Програмування мовою Python (основи, інклюзивний курс). Навчальний посібник підготовлено до друку для самостійної роботи студентів вищих навчальних закладів. – К.: ННІІТ ДУТ, 2022. - 113 с.

Навчальний посібник «Програмування мовою Python» (інклюзивний курс) призначено для вивчення дисципліни «Спеціальні мови програмування» та розроблені відповідно до програми курсу «Спеціальні мови програмування». Предметом вивчення навчальної дисципліни є принцип програмування на мові Python, відповідно до якого студент, користуючись знаннями та навичками у програмуванні на мові Python, розробляє програму за будь-яким завданням, отримує результати чисельно, аналізує код та результати роботи програми. У навчальному посібнику «Програмування мовою Python» (інклюзивний курс) достатньо практичних прикладів для засвоєння теоретичних знань та практичних вмінь для відпрацювання навиків розробки програмного забезпечення мовою Python.

Навчальний посібник може бути використаний студентами галузь знань- "Інформаційні технології" освітньо-кваліфікаційного рівня «бакалавр» для підготовки до лекційних, практичних і лабораторних занять з дисципліни, а також всіма, хто цікавиться питаннями програмування мовою Python.

Зміст

ОПИС НАВЧАЛЬНОЇ ДИСЦИПЛІНИ

Навчальна дисципліна «Спеціальні мови програмування» вивчається студентами освітньо-кваліфікаційного рівня «бакалавр» галузь знань 12, «Інформаційні технології».

Метою викладання навчальної дисципліни є здатність створювати програмне забезпечення для зберігання, видобування та опрацювання даних, реалізовувати фази та ітерації життєвого циклу програмних систем та інформаційних технологій на основі відповідних моделей і підходів розробки програмного забезпечення. Отримання навиків з написання програм для обчислення поставлених задач.

Основними завданнями вивчення дисципліни є формування знань, вмінь та навичок необхідних для виконання випускниками всіх виробничих функцій та типових задач діяльності.

У результаті вивчення навчальної дисципліни студент повинен:

знати: основні процеси, фази та ітерації життєвого циклу програмного забезпечення.

вміти: вибирати та використовувати відповідну задачі методологію створення програмного забезпечення. Вміти застосовувати на практиці ефективні підходи щодо проектування програмного забезпечення.

Навчальний процес організовується шляхом читання лекцій, проведення практичних та лабораторних занять. На лекціях закладаються основи розуміння студентами базових методів тестування та оцінки якості інформаційних систем, на практичних та лабораторних заняттях студенти набувають навички з вибору та використання методів та технологій тестування та оцінки і забезпечення якості урахуванням особливостей задач конкретної інформаційної системи. Для самостійної роботи студентам разом з рекомендованою літературою пропонується користуватися електронними версіями підручників, посібників, документів, що представлені в електронній бібліотеці на сайті університету, а також можливостями мережі Інтернет.

Поточний контроль здійснюється під час проведення практичних та лабораторних занять і має на меті перевірку рівня підготовленості студента до виконання конкретної роботи. Тестовий контроль проводиться на лекційних заняттях. Семестровий контроль проводиться у формі заліку в обсязі навчального матеріалу, визначеного навчальною програмою, і в терміни, встановлені навчальним планом.

Залікова оцінка з дисципліни «Спеціальні мови програмування» виставляється як сума балів, отриманих за виконання лабораторних та практичних робіт, а також балів, які студент отримав за поточні теоретичні опитування.

При самостійній роботі студенти набувають навички самостійного освоєння інструментарію кодування на мові програмування Python, які не використані в навчальному процесі та поглиблюють свої знання щодо мови програмування.

1. ВВЕДЕННЯ У PYTHON

1.1. Основні можливості мови програмування Python та завантаження програми.

Що вміє робити Python:

• Робота с xml/html файлами.

• Робота с http запитами.

• GUI (графічний інтерфейс).

• Створення веб-сценаріїв

• Робота с FTP.

• Робота з зображеннями, аудіо та відео файлами.

• Робототехніка.

• Програмування математичних та наукових обчислень.

Та багато іншого...

Python можно загрузити з веб-сайту https://python.org/downloads/windows/, обираємо python 3.

Щоб написати "**hello world**" на **Python** достатньо усього однієї строки **print** ("Hello world") та натиснемо **Enter.**

Інтерактивний режим не буде основним. В основному ви будете зберігати програмний код в файл і запускати його. Для того, щоб створити нове вікно в інтерактивному режимі **IDLE**, виберіть **File→New File** (або натисніть **Ctrl+N**).

У вікні введіть наступний код: **name=input** ("Як Вас звати?") **Print** ("Привіт,", name). Тепер натиснемо **F5** (або виберемо в меню **IDLE Run→Run Module**) і переконаємося, що програму, яку ми написали, працює. Перед запуском **IDLE** запропонує нам зберегти файл. Збережемо його, де вам буде зручно, після чого програма запуститься.

Перший рядок друкує питання ("Як Вас звати?") і очікує, поки ви не надрукуєте що-небудь і не натиснете **Enter**, і зберігає введене значення в змінній **name**. У другому рядку ми використовуємо функцію **print** для виведення тексту на екран, в даному випадку для виведення "Привіт," і того, що зберігається в змінної "**name**".

1.2. Основи синтаксису Python.

- Кінець рядка є кінцем інструкції (крапка з комою не потрібна).

- Вкладені інструкції об'єднуються в блоки по величині відступів. Відступ може бути будь-яким, головне, щоб в межах одного вкладеного блоку відступ був однаковий. І про читабельність коду не забувайте. Відступ в 1 пробіл, наприклад, не найкраще рішення. Використовуйте 4 пробілу (або знак табуляції).

- Вкладені інструкції в **Python** записуються відповідно одним і тим же шаблоном, коли основна інструкція завершується двокрапкою, слідом за яким розташовується вкладений блок коду, зазвичай з відступом під рядком основної інструкції.

1.3. Введення і виведення даних.

Введення даних здійснюється за допомогою команди **input** (список введення):

```
a = input ()
print (a)
```

У дужках функції можна вказати повідомлення - коментар до даних які вводяться:

```
a = input ("Введіть кількість:")
```

Команда **input ()** за замовчуванням сприймає вхідні дані як рядок символів. Тому, щоб ввести цілочисельне значення, слід вказати тип даних **int ()**:

```
a = int (input ())
```

Для введення дійсних чисел застосовується команда:

```
a = float (input ())
```

Вивід даних здійснюється за допомогою команди **print** (список виведення):

```
a = 1
b = 2
```

```
print (a)
print (a + b)
print ( 'сума =', a + b)
```

Існує можливість запису команд в один рядок, розділяючи їх через **;** . Однак не слід часто використовувати такий спосіб, це знижує читабельність:

```
a = 1; b = 2; print (a)
print (a + b)
print ( 'сума =', a + b)
```

Для команди **print** може додаватися так званий сепаратор - роздільник між елементами виведення:

```
x = 2
y = 5
print (x, "+", y, "=", x + y, sep = "")
```

Результат відобразиться з пробілами між елементами: 2 + 5 = 7.

1.4. Синтаксис команди format у Python.

Для форматованого виведення використовується **format**:

Строковий метод **format ()** повертає відформатовану версію рядка, замінюючи ідентифікатори в фігурних дужках **{}**. Ідентифікатори можуть бути позиційними, числовими індексами, ключами словників, іменами змінних.

Синтаксис команди format:

```
поле замены     ::=  "{" [имя поля] ["!" преобразование] [":" спецификация] "}"
имя поля        ::=  arg_name ("." имя атрибута | "[" индекс "]")*
преобразование  ::=  "r" (внутреннее представление) | "s" (человеческое
↪представление)
спецификация ::=  [[fill]align][sign][#][0][width][,][.precision][type]
заполнитель  ::=  символ кроме '{' или '}'
выравнивание ::=  "<" | ">" | "=" | "^"
знак         ::=  "+" | "-" | " "
ширина       ::=  integer
точность     ::=  integer
```

```
ТИП          ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" |
                  "n" | "o" | "s" | "x" | "X" | "%"
```

Аргументів на **format ()** може бути більше, ніж ідентифікаторів в рядку. В такому випадку ті які залишилися - ігноруються.

Ідентифікатори можуть бути або індексами аргументів, або ключами:

Наприклад: вивід числа 11 без форматування.

>>> x=11

>>> print(x)

Відповідь: 11.

Наприклад: вивід числа 11 з форматуванням.

>>> x=11

>>> print('{:4}'.format(x))

Відповідь: 11.

В результаті виведеться число 11, а перед ним два пробілу, так як було вказано використовувати для виведення чотири знакомісця.

Доступні наступні варіанти вирівнювання:

'<'	Символи-наповнювачі будуть праворуч (вирівнювання об'єкта по лівому краю) (за замовчуванням).
'>'	Вирівнювання об'єкта по правому краю.
'='	Наповнювач буде після знака, але перед цифрами. Працює тільки з числовими типами.
'^'	Вирівнювання по центру.
'>'	Вирівнювання об'єкта по правому краю.

Наприклад:

‘{:>30}’.format(‘Nata’)

Буде 30 знакомісць, а слово Nata буде справа.

'<' Символи-наповнювачі будуть праворуч (вирівнювання об'єкта по лівому краю) (за замовчуванням).

Наприклад:

‘{:<30}’.format(‘Nata’)

Буде 30 знакомісць, а слово Nata буде зліва.

'^' Вирівнювання по центру.

Наприклад:

‘{:^30}’.format(‘Nata’)

Буде 30 знакомісць, а слово Nata буде по центру.

Опція «Знак»

'+'	- Знак має бути надрукований для всіх чисел.
'-'	- Мінус для негативних, нічого для позитивних.
'Пропуск'	- Мінус для негативних, пробіл для позитивних.

'+'	- Знак має бути надрукований для всіх чисел.

Наприклад:

‘{:+f}; {:+f}’.format(3.14, -3.14)

Виведе числа со знаками +3.14; -3.14

'-'	- Мінус для негативних, нічого для позитивних.

Наприклад:

‘{:-f}; {:-f}’.format(3.14, -3.14)

Виведе числа со знаками 3.14; -3.14

'Пропуск'	- Мінус для негативних, пробіл для позитивних.

Наприклад:

‘{: f}; {: f}’.format(3.14, -3.14)

Виведе числа со знаками 3.14; -3.14

Поле «Тип» може приймати наступні значення:

'd','i', 'u'	- Десяткове число.
'o'	- Число в вісімковій системі числення.
'x'	- Число в шістнадцятковій системі числення (букви в нижньому регістрі).
'X'	- Число в шістнадцятковій системі числення (літери у верхньому регістрі).
'e'	- Число з плаваючою точкою з експонентою (експонента в нижньому регістрі).
'E'	- Число з плаваючою точкою з експонентою (експонента в верхньому регістрі).
'f', 'F'	- Число з плаваючою крапкою (звичайний формат).
'g'	- Число з плаваючою крапкою з експонентою (експонента в нижньому регістрі), якщо вона менше, ніж -4 або точності, інакше звичайний формат.
'G'	- Число з плаваючою крапкою. з експонентою (експонента в верхньому регістрі), якщо вона менше, ніж -4 або точності, інакше звичайний формат.
'c'	- Символ (рядок з одного символу або число - код символу).
's'	- Строка.
'%'	- Число множиться на 100, відображається число з плаваючою точкою, а за ним знак %.

Для форматування дійсних чисел з плаваючою точкою використовується наступна команда:

print ('{0: .2f}'. format (дійсне число))

print ('{0: .2f}'. format (дійсне число))

Наприклад:

x=10

y=7

print(‘{0:.4f}’.format(x/y))

Команди виведуть 1.4286 – чотири знаки після коми.

1.5. Типи даних.

- **Цілі числа** (тип **int**) - позитивні і негативні цілі числа, а також 0 (наприклад, 4, 687, -45, 0).

- **Числа з плаваючою точкою** (тип **float**) - дробові, вони ж речові, числа (наприклад, 1.45, -3.789654, 0.00453). *Примітка:* для поділу цілої та дробової частин тут використовується точка, а не кома.

- **Строки** (тип **str**) - набір символів, укладених в лапки (наприклад, "**ball**", "What is your name?", 'dkfjUUv', '6589'). *Примітка:* лапки в **Python** можуть бути одинарними або подвійними; одиночний символ в лапках також є рядком, окремого символьного типу в Пітоні немає.

1.6. Операції в програмуванні.

Операція - це виконання будь-яких дій над даними, які в цьому випадку називають **операндами**. Саму дію виконує оператор - спеціальний інструмент.

Так в математиці і програмуванні символ **плюса** є оператором операції додавання по відношенню до чисел. У разі рядків цей же оператор виконує **операцію конкатенації - з'єднання**.

Наприклад:

>>> 10.25 + 98.36

Відповідь: 108.61

>>> 'Hello' + 'World'

Відповідь: 'HelloWorld'

Дії над цілими числами

^	Зведення у ступінь
*	Множення
/	Ділення
+	Додавання
-	Віднімання
//	Отримання цілої частини від ділення
%	Остача від ділення
-x	Зміна знаку числа
abs	Модуль числа
Divmod(x,y)	Пара (x//y,x%y)
Pow(x,y[,z])	X^y по модулю (якщо модуль надано)

1.7. Системи числення.

- **int ([object], [підставу системи числення])** - перетворення до цілого числа в десятковій системі числення. За замовчуванням система числення десяткова, але можна задати будь-яку підставу від 2 до 36 включно.

- **bin (x)** - перетворення цілого числа в двійкову систему числення.

- **hex (x)** - перетворення цілого числа в шістнадцятирічну систему числення.

- **oct (x)** - перетворення цілого числа в восьмирічну систему числення.

Числа з плаваючою комою підтримують ті ж операції, що і цілі. Однак (через представлення чисел в комп'ютері) речові числа неточні, і це може привести до помилок:

>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1

Виведе 0.9999999999999999

Для високої точності використовують інші об'єкти (наприклад: Decimal і Fraction). Також речові числа не підтримують довгу арифметику.

Наприклад:

```
>>> a = 3 ** 1000
>>> a + 0.1
Traceback (most recent call last):
File "", line 1, in
OverflowError: int too large to convert to float
```

Наприклад:

```
>>> c = 150
>>> d = 12.9
>>> c + d
```

Відповідь: 162.9

Наприклад:

```
>>> p = abs(d - c) # Модуль числа
>>> print(p)
```

Відповідь: 137.1

Наприклад:

```
>>> round(p) # Округлення
```

Відповідь: 137

Додаткові методи роботи з цілими числами:

float.as_integer_ratio ()	-	пара цілих чисел, чиє ставлення одно цього числа.
float.is_integer ()	-	чи є значення цілим числом.
float.hex ()	-	переводить float в hex (шістнадцятирічну

		систему числення).
classmethod **float.fromhex (s)**	-	float з шістнадцяткового рядка

Модуль **math** надає більш складні математичні функції

Наприклад:

>>> import math

>>> math.pi

Відповідь: 3.141592653589793

>>> math.sqrt(85)

Відповідь: 9.219544457292887

Бібліотека модуля math (деякі функції)

math.ceil(x)	Повертає найближче ціле число більше, ніж x
math.fabs(x)	Повертає абсолютне значення числа x
math.factorial(x)	Обчислює факторіал x
math.floor(x)	Повертає найближче ціле число менше, ніж x
math.exp(x)	Обчислює e**x
math.log2(x)	Логарифм по основі 2
math.log10(x)	Логарифм по основі 10
math.log(x[, base])	За замовчуванням обчислює логарифм за основою e, додатково можна вказати основу логарифма
math.pow(x, y)	Обчислює значення x в ступені y
math.sqrt(x)	Корінь квадратний від x

math.ceil(x) - Повертає найближче ціле число більше, ніж x/y

Наприклад:

>>>Import math

>>> x=10

>>> y=7

>>> math.ceil(x/y)

Відповідь: -2.

math.fabs(x) - Повертає абсолютне значення числа x/y

Наприклад:

>>> Import math

>>> x=10

>>> y= -7

>>> math.fabs(x/y)

Відповідь: 1.428...

math.factorial(x) - Обчислює факторіал x

Наприклад:

>>> Import math

>>> x=10

>>> math.factorial(x)

Відповідь: 3628800

math.floor(x) - Повертає найближче ціле число менше, ніж x/y

Наприклад:

>>> Import math

>>> x=10

>>> y= 7

>>> math.floor(x/y)

Відповідь: 1

math.exp(x) - Обчислює e**x

Наприклад:

>>>Import math

>>> x=10

>>> math.exp(x)

Відповідь: 22026.465794806718

math.log2(x) - Логарифм по основі 2

Наприклад:

>>> Import math

>>> x=10

>>> math.log2(x)

Відповідь: 3.321928094887362

math.log10(x) - Логарифм по основі 10

Наприклад:

>>> Import math

>>> x=10

>>> math.log10(x)

Відповідь: 1

math.pow(x, y) - Обчислює значення x в ступені y

Наприклад:

>>> Import math

>>> x=10

>>> y=7

>>> math.pow(x,y)

Відповідь: 10000000.0

math.sqrt(x) - Корінь квадратний від x

Наприклад:

>>> Import math

>>> x=16

>>> math.sqrt(x)

Відповідь 4

Тригометричні функції модуля math

math.cos(x)	повертає cos числа x
math.sin(x)	повертає sin числа x
math.tan(x)	повертає tan числа x
math.acos(x)	повертає acos числа x
math.asin(x)	повертає asin числа x
math.atan(x)	повертає atan числа x

Робота з комплексними числами у прикладах:

Наприклад:

>>> x=complex(1,2)

>>> print(x)

Відповідь: (1+2j)

Наприклад:

>>> y=complex(3,4)

>>> print(y)

Відповідь: (3+4j)

Наприклад:

>>> z=x+y

>>> print(z)

Відповідь: (4+6j)

Наприклад:

>>> z=x*y

>>> print(z)

Відповідь: (-5+10j)

Наприклад:

>>> z=x/y

>>> print(z)

Відповідь: (0.44+0.08j)

Комплексні числа не можна порівнювати!!!

Наприклад:

Отримати сполучене число

>>> print(z.conjugate())

Відповідь: (0.44-0.08j)

Наприклад:

Отримати мниму частину комплексного числа

>>> print(z.imag)

Відповідь: 0.08

Наприклад:

Отримати дійсну частину комплексного числа

>>> print(z.real)

Відповідь: 0.44

Але можливо перевірити нерівність

Наприклад:

>>> print(x==y)

Відповідь: False

Наприклад:

Модуль комплексного числа

>>> abs(x)

Відповідь: 2.23606797749979

Наприклад:

Возведення у ступінь 2

>>> pow(y,2)

Відповідь: (-7+24j)

1.8. Зміна типів даних.

Правила написання імен змінних

- Бажано давати змінним осмислені імена, що говорять про призначення даних, на які вони посилаються.

- Ім’я змінної не повинно збігатися з командами мови (зарезервованими ключовими словами).

- Ім’я змінної має починатися з букви або символу підкреслення (_), але не з цифри.

- Ім’я змінної не повинно містити пробіли.

- Щоб дізнатися значення, на яке посилається змінна, перебуваючи в режимі інтерпретатора, досить її викликати, тобто написати ім'я і натиснути **Enter**.

Для зміни одних типів даних в інші в мові **Python** передбачений ряд вбудованих в нього функцій.

Ці функції перетворять те, що поміщається в їх дужки відповідно в ціле число, дійсне число або рядок.

Наприклад:

Перевести строку 4 у ціле число та до нього додати число 5.

```
>>> int('4')+5
```

Відповідь: 9

Наприклад:

Перевести число 5 у рядок та до нього додати рядок ‘5’.

```
>>> str(5)+'5'
```

Відповідь: '55'

Наприклад:

Перевести дійсне число 5.1, у рядок та до нього додати переведене у рядок число 5.

```
>>> str(5.1)+str(5)
```

Відповідь: '5.15'

Наприклад:

Перевести строку 5.1 у дійсне число, та до нього додати переведене у ціле число 5.3.

```
>>> float('5.1')+int(5.3)
```

Відповідь: 10.1

Ключові питання

1. Результат функції math.sqrt(16)?
2. Результат роботи програми?

```
x = 1//5 + 1/5
print(x)
```

3. Результат роботи програми?

```
x = 1
y = 2
x, y, z = x, x, y
z, y, z = x, y, z
print(x, y, z)
```

4. Результат роботи програми?

```
S1 = 'Sasha'
S2 = '-Masha'
print(S2+S1)
```

5. Результат роботи print(3 * 'abc' + 'xyz')?
6. Результат роботи print(float('1, 3'))?
7. Результат роботи програми?

```
x = float(input())
y = float(input())
print(y**(1/x))
```

8. Результат роботи програми?

```
x = int(input())
y = int(input())
x = x % y
```

```
x = x % y
y = y % x
print(y)
```

Завдання на самостійну роботу

1. Змінній var_int надайте значення 10, var_float - значення 8.4, var_str - "No".
2. Змініть значення, збережене в змінної var_int, збільшивши його в 3.5 рази, результат зв'яжіть зі змінною var_big.
3. Змініть значення, збережене в змінної var_float, зменшивши його на одиницю, результат зв'яжіть з тієї ж змінної.
4. Розділіть var_int на var_float, а потім var_big на var_float.
5. Результат даних виразів не прив'язуйте до жодних змінних.
6. Виведіть значення всіх змінних.
7. Напишіть програму, яка запитувала б у користувача:
   - ПІБ ("Ваші прізвище, ім'я, по батькові?")
   - вік ("Скільки Вам років?")
   - місце проживання ("Де Ви живете?")
   - де Ви навчаєтесь (“Де Ви навчаєтесь?”)
   - номер Вашої групи(“Номер Вашої групи?”)
   - порядковий номер по списку у групі(“Який Ваш порядковий номер у списку групи?”)

- ***питання відповідно до варіанту***

Після цього виводила б рядки:

"Ваше ім'я"

"Ваш вік"

"Ви живете в"

"Ви навчаєтесь в".

"Номер моєї групи -"

"Мій порядковий номер у списку групи-"

«Ваш варіант відповіді»

№ Варіанту - остання цифра у списку групи.

№ варіанту	Питання
0	Як справи?
1	Як Ви себе почуваєте?
2	Коли будете вдома?
3	Яку оцінку отримав на ЗНО по українській мові?
4	Сьогодні сонячно?
5	Коли нарешті карантин?
6	Як звати Вашого друга?
7	Ви думаєте вступати у магістратуру?
8	Якого кольору Ваш зошит?
9	Який Ваш настрій сьогодні?

8. Написати програму для розрахунку Z

$$Z = \frac{9\pi t + 10\cos(x)}{\sqrt{t} - |\sin(t)|} * e^x$$

Для введення даних використовуйте команду **input**, визначивши тип змінних.

Результат вивести з двома знаками після коми. x - остання цифра у списку групи, t=1.

2. ЦИКЛИ У PYTHON

2.1. Інструкція if-elif-else у Python.

Умовна інструкція **if-elif-else** (її ще іноді називають оператором розгалуження) - основний інструмент вибору в **Python**. Простіше кажучи, вона

вибирає, яку дію слід виконати, в залежності від значення змінних в момент перевірки умови.

Синтаксис інструкції if

Спочатку записується частина **if** з умовним виразом, далі можуть слідувати одна або більше необов'язкових частин **elif**, і, нарешті, необов'язкова частина **else**. Загальна форма записи умовної інструкції **if** виглядає наступним чином:

```
if test1:
        state1
elif test2:
        state2
else:
        state3
```

Наприклад:

```
a = int(input())

if a < -5:

       print('Low')

elif -5 <= a <= 5:
        print('Mid')
else:
        print('High')
```

Перевірка істинності в Python

- Будь-яке число, не рівне 0, або непорожній об'єкт - істина.
- Числа, рівні 0, порожні об'єкти і значення **None** – брехня.
- Операції порівняння застосовуються до структур даних рекурсивно
- Операції порівняння повертають **True** або **False**

- Логічні оператори **and** і **or** повертають істинний або помилковий об'єкт-операнд

Логічні оператори:

X and Y

Істина, якщо обидва значення X і Y істинні.

X or Y

Істина, якщо хоча б одне зі значень X або Y істинно.

not X

Істина, якщо X хибно.

Тримісний вираз if / else

Наступна інструкція:

if X:

A = Y

else:

A = Z

Цей вираз займає цілих 4 рядки. Спеціально для таких випадків і був вигадан вираз **if / else**:

A = Y if X else Z

У даній інструкції інтерпретатор виконає вираз Y, якщо X істинно, в іншому випадку виконається вираз Z.

>>> A = 't' if 'spam' else 'f'

>>> A

Відповідь: 't'

2.2. Цикл while у Python.

Виконує тіло циклу до тих пір, поки умова циклу істино.

Наприклад:

```
>>> i = 5
>>> while i < 15:
... print(i)
... i = i + 2
```

Відповідь:

5

7

9

11

13

Наприклад:

Написати програму в якій число циклів ми вводимо за допомогою команди **input** з коментаріями "Введіть кількість циклів:" , числа вводимо за допомогою команди **input** з клавіатури, та числа виводимо з коментарями “Менше 5”, “від 5 до 10”, “Більше 10”.

```
a = int(input("Введіть кількість циклів:"))
i = 1
while i <= a:
c = int(input(‘Введить число:’))
if c < 5:
        print(' Менше 5',c)
         i=i+1
elif 5 <= c <= 10:
         print(' від 5 до 10',c)
         i=i+1
else:
        print(' Більше 10',c)
        i=i+1
```

Наприклад:

Вводиться з клавіатури ціле число, що не менше 2. Виведіть його найменший натуральний дільник, відмінний від 1 Вирішити задачу використовуючи циклічну конструкцію **while**. (У циклі **while** в якості логічного виразу використовується команда n% і порівнювана з нулем.)

```
a = int(input("Введіть ціле число:"))
i = 2

while a % i !=0:

    i=i+1

print('Найменшій натуральний дільник',i)
```

2.3. Цикл for у Python.

Цикл **for** вже трішки складніший, трохи менш універсальний, але виконується набагато швидше циклу **while**. Цей цикл проходиться по будь-якому ітерованому об'єкту (наприклад: рядку або списку), і під час кожного проходу виконує тіло циклу.

Наприклад:

```
for i in 'nata':

        print(i*3, end='')
```

Відповідь: nnnaaatttaaa

Без end = '' – розташовується у стовпчик.

Наприклад:

```
for i in 'nata':
    print(i*3)
```

Відповідь:

nnn

aaa

ttt

aaa

Оператор break

Достроково перериває цикл.

Наприклад:

```
for i in 'nata':
    if i=='t':
        break
    print(i*3)
```

Відповідь:

nnn

aaa

Оператор continue

Починає наступний прохід циклу, минаючи решту тіла циклу (**for** або **while**).

Наприклад:

```
for i in 'nata':
    if i==’t’:
        continue
    print(i*3)
```

Відповідь:

```
nnn
aaa
aaa
```

Слово **else**, застосоване в циклі **for** або **while**, перевіряє, чи був проведений вихід з циклу інструкцією **break**, або ж "природним" чином.

Наприклад:

```
for i in 'nata':
    if i==’j’:
        break
    else:
        print(‘Букви j нема’)
```

Відповідь:

```
Букви j нема
Букви j нема
Букви j нема
```

Букви j нема

Ключові питання

1. Скільки зірочок нарахує програма?

```
i = 0
while i < i + 2:
    i += 1
    print('*')
else:
    print('*')
```

2. Скільки елементів у списку s = [i for i in range (-1, 2)]?

3. Результат роботи програми ?

```
dct = {'one':'two', 'three':'one',  'two':'three'}
v = dct['three']
for k in range(len(dct)):
    v = dct[v]
print(v)
```

4. Визначити, що виводить код, поданий нижче?

```
for i in 'bowling':
    if i == 'i':
        break
    print (i * 2, end='')
```

5. Визначити, що виводить код, поданий нижче?

```
for i in 'bowling':
    if i == 'i':
        continue
    print (i * 2, end='')
```

6. Скільки зірочок виведе програма?

```
for i in range(5):
    if i == 4:
```

```
        continue
    print("*")
```

7. Скільки зірочок виведе програма?

```
for i in range(6):
    if i == 2:
        break
    print("*")
```

Завдання на самостійну роботу

1. Знайти суму **n** елементів наступного ряду чисел: 1 -0.5 0.25 -0.125 ... n. Кількість елементів (**n**) вводиться з клавіатури. Вивести на екран кожен член ряду і його суму. Вирішити задачу використовуючи циклічну конструкцію **for**.

2. Дано ціле число, що не менше 2. Виведіть його найменший натуральний дільник, відмінний від 1 Вирішити задачу використовуючи циклічну конструкцію **while**. (У циклі **while** в якості логічного виразу використовується команда n% і порівнювана з нулем.)

3. Написати програму. Програма має вивести ряд послідовності Фібоначчі. Кожен наступний член дорівнюється сумі двох попередніх.

0, 1, 1, 2, 3, 5, 8, 13……..

4. “Вгадайте число від 1 до 20”

4.1. По запрошенню “ Введіть число від 0 до 20” вводиться число.

4.2. Генератор випадкових чисел генерує послідовність від 0 до 20.

4.3. Якщо запропоноване число менше генерованого виведіть “Запропоноване число менше загаданого”.

4.4. Якщо запропоноване число більше генерованого виведіть “Запропоноване число більше загаданого”.

4.5. Якщо числа співпадають виводиться на якому кроці числа співпали та коментарій “Ви вгадали число з ….. разу ”.

4.6. Якщо Ви ввели число не в діапазоні 0-20 вивести “Невірне значення!!!”

3. РЯДКИ

Рядки в апострофах і в лапках - одне і те ж. Причина наявності двох варіантів в тому, щоб дозволити вставляти в літерали рядків символи лапок і апострофів, не використовуючи екранування. Екрановані послідовності дозволяють вставити символи, які складно ввести з клавіатури.

3.1 Функції та методи рядків.

Функція або метод	Призначення
S = ‘str’; S = “str”; S = ‘’‘str’’’; **S = “””str”””**	Літерали рядків
S = “s\np\ta\nbbb”	Екрановані послідовності
S = r”C:\temp\new”	Неформатовані рядки (пригнічують екранування)
S = b”byte”	Рядок байтів
S1 + S2	Конкатенація (додавання рядків)
S1 * 3	Повторення рядка
S[i]	Звернення за індексом
S[i:j:step]	Витяг зрізу
len(S)	Довжина рядка
S.find(str, [start],[end])	Пошук підрядка в рядку. Повертає номер першого входження або -1
S.rfind(str, [start],[end])	Пошук підрядка в рядку. Повертає номер останнього входження або -1
S.index(str, [start],[end])	Пошук підрядка в рядку. повертає номер першого входження або викликає ValueError
S.rindex(str, [start],[end])	Пошук підрядка в рядку. Повертає номер останнього входження або викликає ValueError
S.replace(шаблон, заміна)	Заміна шаблону
S.split(символ)	Розбиття рядка по роздільнику
S.isdigit()	Чи полягає рядок з цифр
S.isalpha()	Чи полягає рядок з букв

Функція або метод	Призначення
S.isalnum()	Чи полягає рядок з цифр або букв
S.islower()	Чи полягає рядок із символів в нижньому регістрі
S.isupper()	Чи полягає рядок із символів у верхньому регістрі
S.isspace()	Чи має рядок символи, що не відображаються (пробіл, символ перекладу сторінки ('\ f'), "новий рядок" ('\ n'), "переклад каретки" ('\ r'), "горизонтальна табуляція" ('\ t') і "вертикальна табуляція " ('\ v'))
S.istitle()	Чи починаються слова в рядку з великої літери
S.upper()	Перетворення рядка до верхнього регістру
S.lower()	Перетворення рядка до нижнього регістру
S.startswith(str)	Чи починається рядок S з шаблону str
S.endswith(str)	Закінчується рядок S шаблоном str
S.join(список)	Збірка рядка зі списку з роздільником S
ord(символ)	Символ в його код ASCII
chr(число)	Код ASCII в символ
S.capitalize()	Змінює перший символ рядка в верхній регістр, а всі інші в нижній
S.center(width, [fill])	Повертає відцентрований рядок, по краях якого стоїть символ **fill** (пробіл за замовчуванням)
S.count(str, [start],[end])	Повертає кількість непересічних входжень підрядка в діапазоні [початок, кінець] (0 і довжина рядка за замовчуванням)
S.expandtabs([tabsize])	Повертає копію рядка, в якій всі символи табуляції замінюються одним або декількома пропусками, в залежності від поточного стовпця. Якщо TabSize НЕ вказано, розмір табуляції вважається рівним 8 прогалин
S.lstrip([chars])	Видалення символів пробілів на початку рядка
S.rstrip([chars])	Видалення символів пробілів у кінці рядку
S.strip([chars])	Видалення символів пробілів на початку

Функція або метод	Призначення
	і в кінці рядка
S.partition(шаблон)	Повертає кортеж, що містить частину перед першим шаблоном, сам шаблон, і частину після шаблону. Якщо шаблон не знайдений, повертається кортеж, що містить сам рядок, а потім дві порожніх рядки
S.rpartition(sep)	Повертає кортеж, що містить частину перед останнім шаблоном, сам шаблон, і частину після шаблону. Якщо шаблон не знайдений, повертається кортеж, що містить дві порожні рядки, а потім сам рядок
S.swapcase()	Перекладає символи нижнього регістра в верхній, а верхнього - в нижній
S.title()	Першу букву кожного слова переводить в верхній регістр, а всі інші в нижній
S.zfill(width)	Робить довжину рядку не меншою width, в разі потреби заповнюючи перші символи нулями
S.ljust(width, fillchar=” ”)	Робить довжину рядку не меншою width, в разі потреби заповнюючи останні символи символом fillchar
S.rjust(width, fillchar=” ”)	Робить довжину рядку не меншою width, в разі потреби заповнюючи перші символи символом fillchar
S.format(*args, **kwargs)	форматування рядка

Наприклад:

Конкатенація (додавання)

```
>>> S1 = 'spam'
>>> S2 = 'eggs'
>>> print(S1 + S2)
```

Відповідь: 'spameggs'

Наприклад:

Дублювання рядка

>>> print('spam' * 3)

Відповідь: spamspamspam

Наприклад:

Доступ до індексу

>>> S = 'spam'

>>> S[0]

Відповідь: 's'

>>> S[2]

Відповідь: 'a'

>>> S[-2]

Відповідь: 'a'

Як видно з прикладу, в **Python** є можливість доступу по негативному індексу, при цьому відлік йде від кінця рядка.

Витяг зрізу. Оператор вилучення зрізу: [X: Y]. X - початок зрізу, а Y - закінчення; символ з номером Y в зріз не входить. За замовчуванням перший індекс дорівнює 0, а другий - довжині рядка.

Наприклад:

>>> s = 'spameggs'

>>> s[3:5]

Відповідь: 'me'

>>> s[2:-2]

Відповідь: 'ameg'

>>> s[:6]

Відповідь: 'spameg'

>>> s[1:]

Відповідь: 'pameggs'

>>> s[:]

Відповідь: 'spameggs‘

Можливо задати крок з яким треба витягувати зріз.

Наприклад:

>>> s[::-1]

Відповідь: 'sggemaps'

>>> s[3:5:-1]

Відповідь: ''

>>> s[2::2]

Відповідь: 'aeg'

Ключові питання

1. Результат роботи програми?

S = 'Masha'

S[0]

2. Результат роботи програми?

S = 'Masha-Sasha'

S[3:5]

3. Результат роботи програми?

S = 'MashaSasha'

S[:]

4. Результат роботи програми?

S = 'Masha-Sasha'

S[::-1]

5. Результат роботи програми?

S = 'Masha-Sasha'

S[2::2]

6. Результат роботи програми?

S = 'Masha-Sasha'

S[2:5:-1]

7. Результат роботи програми?

S = 'Masha-Sasha'

S[-2::-1]

Завдання на самостійну роботу

1. Перетворення рядка до верхнього регістру згідно до варіанту**.**

2. Перетворення рядка до нижнього регістру.

3. Переведіть першу букву кожного слова в верхній регістр, а всі інші в нижній.

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

№ Варіант - остання цифра у списку групи.

№	Рядок
1	впродовж доби в Україні захворіли на коронавірус
2	до лікарень госпіталізували
3	в Україні провели 68 569 тестів на коронавірус
4	відтепер підприємці з річним доходом до 50 мільйонів гривень зможуть отримати пільгові кредити на суму до 1 мільйона гривень за ставкою 12% річних
5	умови для отримання іпотеки під 7% спростять, а саме змінять вимоги до об'єкта нерухомості

№	Рядок
6	що відомо про програму "5–7–9"
7	ставка кредиту становить 5, 7 та 9% в залежності від обсягу річного доходу та кількості нових робочих місць
8	генеральний директор Apple Тім Кук опублікував на своїй сторінці у Twitter пост про відкриття у Стамбулі нового фірмового магазину компанії
9	фірмові процесори M1 Pro і M1 Max, які здивували своєю продуктивністю
0	виготовлена "з м'якого неабразивного матеріалу" і підходить для очищення дисплеїв пристроїв Apple, у тому числі зі скла з нанотекстурою

1. Присвоїть s1= рядок «Зараховано».
2. Присвоїть s2= рядок «Сенсація».
3. Присвоїть s3= рядок «Сенсація* Сенсація* Сенсація».
4. Присвоїть s4= рядок «ОхОхОхАх».
5. За допомогою команди print вивести значення s1= Зараховано, s2= Сенсація, s3= Сенсація* Сенсація* Сенсація, s4=ОхОхОхАх

   Вивести суму рядків s1 та s2.
6. Повторити рядок s1 чотири рази.
7. Вивести елемент рядку s1 з індексом 3.
8. Витяг зрізу рядку s1 починаючи з індексу 2 та завершуючи індексом 4.
9. Дізнатись кількість входжень підрядка s2 у рядок s3. Результат вивести на екран.

4. СПИСКИ

Списки в **Python** - впорядковані змінювані колекції об'єктів довільних типів

(Майже як масив, але типи можуть відрізнятися).

Щоб використовувати списки, їх потрібно створити. Створити список можна декількома способами. Наприклад, можна обробити будь-який ітеруемий об'єкт (наприклад, рядок) вбудованою функцією **list:**

```
>>> list('123456789')
```

['1', '2', '3', '4', '5', '6', '7', '8', '9']

Пустий список s=[]

>>> s=[]

>>> print(s)

Відповідь: []

Як видно з прикладу, список може містити будь-яку кількість будь-яких об'єктів (у тому числі і вкладені списки), чи не містити нічого.

4.1. Генератори списків.

І ще один спосіб створити список - це генератори списків. Генератор списків – спосіб побудувати новий список, застосовуючи вираз до кожного елементу послідовності.

Генератори списків дуже схожі на цикл **for**.

Наприклад:

>>> c = [c * 3 for c in 'list']

>>> c

Відповідь: ['lll', 'iii', 'sss', 'ttt']

Наприклад: виділить цифру 4 зі списку:

>>> s=[s*2 for s in'0123456789'if s!='4']

>>> s

Відповідь: ['00', '11', '22', '33', '55', '66', '77', '88', '99']

Спосіб - складення однакових списків замінюється множенням. Список з 10 елементів заповнених одиницями

l=[l]*10

print('l=',l)

Відповідь l=[1,1,1,1,1,1,1,1,1,1]

Спосіб

```
l = [i for i in range(10)]
print('l=',l)
```

Відповідь l=[0,1,2,3,4,5,6,7,8,9]

Модуль random надає функції для генерації випадкових чисел, букв, випадкового вибору елементів послідовності.

random.randint (A, B) - випадкове ціле число N, $A \leq N \leq B$.

random.random () - випадкове число від 0 до 1.

Випадкові числа в списку:

10 чисел, генерованих випадковим чином в діапазоні (10,80)

```
from random import randint
l = [randint (10,80) for x in range (10)]
```

10 чисел, генерованих випадковим чином в діапазоні (0,1)

```
l = [random () for i in range (10)]
```

Наприклад:

Десять чисел генерованих випадковим чином у інтервалі 10-80.

```
>>> from random import *
>>> l=[randint(10,80) for i in range(10)]
>>> l
```

Відповідь: [74, 67, 30, 59, 17, 19, 48, 40, 73, 36]

Наприклад:

Десять чисел генерованих випадковим чином у інтервалі 0-1.

```
>>> from random import*
>>> l=[random() for i in range(10)]
>>> l
```

Відповідь: [0.5427014818182992, 0.5216190256125626, 0.8488009894524522, 0.9356589629898409, 0.4745772062771666, 0.9015821164130001, 0.507633154338545, 0.7772208390744064, 0.4060627051380439, 0.8952233049263967]

4.2. Функції та методи списку.

Метод	Що робить
list.append(x)	Додає елемент в кінець списку
list.extend(L)	Розширює список list, додаючи в кінець все елементи списку L
list.insert(i, x)	Вставляє на і-ий елемент значення x
list.remove(x)	Видаляє перший елемент у списку, який має значення x. ValueError, якщо такого елемента не існує.
list.pop(i)	Видаляє і-ий елемент і повертає його. Якщо індекс не вказано, видаляється останній елемент
list.index(x,[start [, end]])	Повертає положення першого елемента зі значенням x (при цьому пошук ведеться від start до end)
list.count(x)	Повертає кількість елементів зі значенням x
list.sort([key=функція])	Сортує список на основі функції
list.reverse()	Розгортає список
list.copy()	Поверхнева копія списку
list.clear()	Очищає список

Списки можна додавати за допомогою знака **«+»**

Створення списку за допомогою функції **Split ().**

Використовуючи функцію **split** в **Python** можна отримати з рядка список.

```
stroka = "Привіт, країна"
lst = stroka.split (",")
```

4.3. Введення, виведення списку.

Для введення елементів списку використовується цикл **for** і команда **range ()**:

```
for i in range (N):
    x [i] = int (input ())
```

Простіший варіант введення списку:

```
x = [int (input ()) for i in range (N)]
```

Вивід цілого списку (масиву):

```
print (L)
```

Поелементний вивід списку :

```
for i in range(N):
        print ( L[i], end = " " )
```

Першій спосіб. Наприклад:

```
print('Ввести список')
x=[]
for i in range(4):
        x.append(int(input()))
print(x)
```

Результат роботи:

```
Ввести список
3
3
4
5
[3,3,4,5]
```

Другий спосіб. Наприклад:

```
x=[]
print('Ввести список')
x=[int(input()) for i in range(4)]
```

```
print(x)
```

Результат роботи:

```
Ввести список
1
2
3
4
[1,2,3,4]
```

Взяття елемента за індексом

```
>>> a = [1, 3, 8, 7]
>>> a[0]
1
>>> a[3]
7
>>> a[4]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Як і в багатьох інших мовах, нумерація елементів починається з нуля. При спробі доступу до неіснуючого індексу виникає виняток **IndexError**. В даному прикладі змінна a була списком, однак взяти елемент за індексом можна і у інших типів: рядків, кортежів.

В **Python** також підтримуються негативні індекси, при цьому нумерація йде з кінця, наприклад:

```
>>> a = [1, 3, 8, 7]
>>> a[-1]
```

Відповідь: 7

```
>>> a[-4]
```

Відповідь: 1

>>> a[-5]

Traceback (most recent call last):

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

IndexError: list index out of range

В **Python**, крім індексів, існують ще й зрізи.

item [START: STOP: STEP] - бере зріз від номера START, до STOP (не включаючи його), з кроком STEP.

За замовчуванням START = 0, STOP = довжина об'єкта, STEP = 1. Відповідно, якісь (а можливо, і всі) параметри можуть бути опущені.

Наприклад:

>>> a = [1, 3, 8, 7]

>>> a[:]

Відповідь: [1, 3, 8, 7]

>>> a[1:]

Відповідь: [3, 8, 7]

>>> a[:3]

Відповідь: [1, 3, 8]

>>> a[::2]

Відповідь: [1, 8]

Усі ці параметри можуть бути негативними.

Наприклад:

>>> a = [1, 3, 8, 7]

>>> a[::-1]

Відповідь: [7, 8, 3, 1]

>>> a[:-2]

Відповідь: [1, 3]

>>> a[-2::-1]

Відповідь: [8, 3, 1]

>>> a[1:4:-1]

Відповідь: []

В останньому прикладі вийшов порожній список, так як START <STOP, а STEP негативний. Те ж саме відбудеться, якщо діапазон значень виявиться за межами об'єкта:

Наприклад:

>>> a = [1, 3, 8, 7]

>>> a[10:20]

Відповідь: []

Також за допомогою зрізів можна не тільки отримувати елементи, але і додавати і видаляти елементи (зрозуміло, тільки для змінних послідовностей).

Наприклад:

>>> a = [1, 3, 8, 7]

>>> a[1:3] = [0, 0, 0]

>>> a

Відповідь: [1, 0, 0, 0, 7]

>>> del a[:-3]

>>> a

Відповідь: [0, 0, 7]

Ключові питання

1. Скільки елементів у списку s = [i for i in range (-1, 2)]?

2. Результат роботи програми?

S = [3, 1, -1]

S [-1] = S [-2]

```
print(S)
```

3. Результат роботи програми?

```
lst = [1, 2]
for v in range(2):
    lst.insert(-1, lst[v])
print(lst)
```

Завдання на самостійну роботу

1. Згенеруйте 10+№ (де №-остання цифра студента у списку групи) випадкових чисел у діапазоні (30, 90).

2. Заповніть список квадратами чисел від 0 до 9, використовуючи генератор списку.

3. Заповніть список з 10+№ (де №-остання цифра студента у списку групи) числами, де кожне наступне число більше на 2.

4. Створіть будь-який список.

а.) Розширте список , додавши до нього усі елементи списку L[3, 6, 7].

б.) Вставте на другий елемент значення 33333.

в.) Розташуйте список в зворотньому порядку.

г.) У кінець списку додайте 3.

д.) Видаліть перший елемент списку який має значення 3.

е.) Розташуйте список у порядку збільшення.

ж.) Очистить список.

5. Створіть будь-який список з 10 цілих чисел.

а.) Виведіть другий та шостий та четвертий елемент списку.

б.) Зрізати по одному символу з начала та кінцю списку.

6. У випадково сгенерованому списку від 1 до 50, який створено з 20 чисел, змініть усі числа, більше шостого числа, на середнє арифметичне усіх чисел списку.

7. У випадково сгенерованому списку від -10 до 10, який створено з 20 чисел, створити новий список у котрому спершу йдуть від’ємні числа, потім нулі, потім позитивні числа.

5. КОРТЕЖ

Кортеж - не змінний!! Інформація захищена від змін. Має менший розмір. Використовують як словник.

Наприклад:

```
>>> a = (1, 2, 3, 4, 5, 6)
>>> b = [1, 2, 3, 4, 5, 6]
>>> a.__sizeof__()
```

Відповідь: 36

```
>>> b.__sizeof__()
```

Відповідь: 44

Наприклад:

Зробимо пустий кортеж.

```
>>> a = tuple() # За допомогою вбудованої функції tuple()
>>> a
```

Відповідь: ()

```
>>> a = () # За допомогою літерала кортежу
>>> a
```

Відповідь: ()

```
>>>
```

Наприклад:

Зробимо кортеж з одного елементу:

```
>>> a = ('s', )
>>> a
```

Відповідь: ('s',)

Або

>>> a = 's',

>>> a

Відповідь: ('s',)

Кортеж з ітерованого об'єкта можна за допомогою **функції tuple ()**

Наприклад:

>>> a = tuple('hello, world!')

>>> a

Відповідь: ('h', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!')

Всі операції над списками, що не змінюють список (додавання, множення на число, методи **index ()** і **count ()** і деякі інші операції). Можна також по-різному змінювати елементи місцями і так далі.

Ключові питання

1. Припускаючи, що **tuple** є правильно створеним кортежем, той факт, що кортежі не змінюється, означає що наступна команда tuple[1] = tuple[1] + tupple[0]:

А. Не правильна

Б. Повністю правильна

В. Може бути виконана якщо кортеж має два та більше елементів.

2. Результат роботи програми?

```
tup = (1, 2, 4, 8)
tup = tup[-2: -1]
tup = tup[-1]
print(tup)
```

Завдання на самостійну роботу

1. Знайдіть добуток цифр у тризначному числі, яке введено користувачем.

Дані натуральні числа від 35 до 87. Програма має вивести числа, які при діленні на 7 дають остачу 1, 2, 5.

2. В масиві у порядку зменшення є купюри [1000, 500, 200, 100, 50, 20, 10, 5, 2, 1]. Реалізувати видачу суми, введеної з екрану мінімальною кількістю купюр.

3. Дано два масиви. Зробити третій масив з двох цих масивів. Розташування у масиві у порядку зростання.

6. СЛОВНИКИ

Словники в **Python** - невпорядковані колекції довільних об'єктів з доступом по ключу. Їх іноді ще називають асоціативними масивами або хеш-таблицями. Щоб працювати зі словником, його потрібно створити. Створити його можна кількома способами.

1. **По-перше**, за допомогою літерала:

Наприклад:

```
>>> d = {}
>>> d
{}
>>> d = {'dict': 1, 'dictionary': 2}
>>> d
```

Відповідь: {'dict': 1, 'dictionary': 2}

2. **По-друге**, за допомогою функції **dict**:

Наприклад:

```
>>> d = dict(short='dict', long='dictionary')
>>> d
```

{'short': 'dict', 'long': 'dictionary'}

>>> d = dict([(1, 1), (2, 4)])

>>> d

Відповідь: {1: 1, 2: 4}

3. **По-третє**, за допомогою методу fromkeys:

Наприклад:

>>> d = dict.fromkeys(['a', 'b'])

>>> d

Відповідь: {'a': None, 'b': None}

>>> d = dict.fromkeys(['a', 'b'], 100)

>>> d

Відповідь: {'a': 100, 'b': 100}

4. **По-четверте**, за допомогою генераторів словників, які дуже схожі на генератори списків.

Наприклад:

>>> d = {a: a ** 2 for a in range(7)}

>>> d

Відповідь: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}

6.1. Методи словників.

dict.clear ()	-	очищає словник
dict.copy ()	-	повертає копію словника
classmethod dict.fromkeys (seq [, value])	-	створює словник з ключами з seq і значенням value (за замовчуванням None)
dict.get (key [, default])	-	повертає значення ключа, але якщо його немає, не кидає виняток, а повертає default (за

		замовчуванням None)
dict.items ()	-	повертає пари (ключ, значення)
dict.keys ()	-	повертає ключі в словнику
dict.pop (key [, default])	-	видаляє ключ і повертає значення. Якщо ключа немає, повертає default (за замовчуванням кидає виняток).
dict.popitem ()	-	видаляє і повертає пару (ключ, значення). Якщо словник порожній, кидає виняток KeyError. Пам'ятайте, що словники невпорядковані
dict.setdefault (key [, default])	-	повертає значення ключа, але якщо його немає, не кидає виняток, а створює ключ з значенням default (за замовчуванням None)
dict.update ([other])	-	оновлює словник, додаючи пари (ключ, значення) з other. Існуючі ключі перезаписуються. Повертає None (не новий словник!).
dict.values ()	-	повертає значення у словнику

Зі словниками можна робити теж саме, що і з іншими об'єктами: вбудовані функції, ключові слова (наприклад, цикли **for** і **while**), а також спеціальні методи словників.

7. МНОЖИНА

Множина в **python** - "контейнер", що містить елементи у випадковому порядку які не повторюються.

Створюємо множину:

Наприклад:

>>> a = set('hello')

>>> a

Відповідь: {'h', 'o', 'l', 'e'}

>>> a = {'a', 'b', 'c', 'd'}

>>> a

Відповідь: {'b', 'c', 'a', 'd'}

Множину використовують для видалення повторюваних елементів.

Наприклад:

>>> words = ['hello', 'daddy', 'hello', 'mum']

>>> set(words)

Відповідь: {'hello', 'daddy', 'mum'}

З множинами можна виконувати безліч операцій: знаходити об'єднання, перетин.

len (s)	-	число елементів у множині (розмір множини)
x in s	-	чи належить x множині s
set.isdisjoint (other)	-	істина, якщо set і other не мають спільних елементів
set == other	-	все елементи set належать other, все елементи other належать set
set.issubset (other) або set <= other	-	все елементи set належать other
set.issuperset (other) або set> = other	-	аналогічно
set.union (other, ...) або set | other | ...	-	об'єднання декількох

		множин
set.intersection (other, ...) або set & other & ...	-	перетин
set.difference (other, ...) або set - other - ...	-	безліч з усіх елементів set, які не належать жодному з other
set.symmetric_difference (other); set ^ other	-	безліч з елементів, що зустрічаються в одному безлічі, але не зустрічаються в обох
set.copy ()	-	копія безлічі

Наприклад:

>>> a=set('forex')

>>> a

Відповідь: {'f', 'x', 'e', 'o', 'r'}

>>> d=set('name')

>>> d

Відповідь: {'a', 'm', 'n', 'e'}

>>> a|d

Відповідь: {'f', 'x', 'a', 'n', 'e', 'o', 'm', 'r'}

>>> a&d

Відповідь: {'e'}

І операції, безпосередньо змінюють множини:

• set.update (other, ...); set | = other | ... - об'єднання.

• set.intersection_update (other, ...); set & = other & ... - перетин.

• set.difference_update (other, ...); set - = other | ... - віднімання.

• set.symmetric_difference_update (other); set ^ = other - безліч з елементів, що зустрічаються в одному безлічі, але не зустрічаються в обох.

• set.add (elem) - додає елемент в множину.

• set.remove (elem) - видаляє елемент з безлічі. KeyError, якщо такого елемента не існує.

• set.discard (elem) - видаляє елемент, якщо він знаходиться в множині.

• set.pop () - видаляє перший елемент з множини. Так як множина не впорядковані, не можна точно сказати, який елемент буде першим.

• set.clear () - очищення множини.

Наприклад:

```
>>> a=set('forex')
>>> a
Відповідь: {'f', 'x', 'e', 'o', 'r'}
>>> a.add('y')
>>> a
Відповідь: {'f', 'x', 'e', 'o', 'y', 'r'}
>>> a.remove('e')
>>> a
Відповідь: {'f', 'x', 'o', 'y', 'r'}
```

Єдина відмінність set від frozenset полягає в тому, що set - змінюваний тип даних, а frozenset - ні.

Наприклад:

```
>>> a = set('qwerty')
>>> b = frozenset('qwerty')
>>> a == b
Відповідь: True
>>> True
True
>>> type(a - b)
<class 'set'>
```

```
>>> type(a | b)
<class 'set'>
>>> a.add(1)
>>> b.add(1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Ключові питання

1. Чим відрізняється кортеж від списку?
2. За допомогою якої вбудованої функції зробити пустий кортеж?
3. Як зробити кортеж з одного елементу?
4. Результат роботи програми?

```
dct = {'one':'two', 'three':'one',  'two':'three'}
v = dct['three']
for k in range(len(dct)):
    v = dct[v]
print(v)
```

Завдання на самостійну роботу

1. Створити будь яку множину .
2. З v = ['ccc', 'ddd', 'yyy', 'iii', ‘ccc’, ‘dd’] видалити елементи які повторюються.
3. Зробить копія множини v = ['ccc', 'ddd', 'yyy', 'iii', ‘ccc’, ‘dd’]
4. Розрахувати кількість елементів у множині v = ['ccc', 'ddd', 'yyy', 'iii', ‘ccc’, ‘dd’].
5. Створіть дві множини . Об’єднайте їх. Знайдіть їх перетин.
6. Додайте будь який елемент у множину.
7. Виділіть будь який елемент із множини.

8. Видаліть перший елемент із множини.

9. Очистіть усі множини.

8. РОБОТА З МАТРИЦЯМИ

Матрицями називаються масиви елементів, які представлені у вигляді прямокутних таблиць. Для матриць визначені правила математичних дій. Елементами матриці можуть бути числа, алгебраїчні символи або математичні функції.

Для роботи з матрицями в **Python** також використовуються списки. Кожен елемент списку-матриці містить вкладений список. Таким чином, виходить структура з вкладених списків. Кількість вкладених списків визначає кількість стовпців матриці, а число елементів всередині кожного вкладеного списку вказує на кількість рядків у вихідній матриці.

Використаємо введення списку (масиву) в мові **Python**.

Для введення елементів списку використовується цикл for і команда range ():

```
for i in range (N):
    x [i] = int (input ())
```

Простіший варіант введення списку:

```
x = [int (input ()) for i in range (N)]
```

Функція int тут використовується для того, щоб рядок, введений користувачем, перетворювався у цілі числа.

Вивід цілого списку (масиву):

```
print (L)
```

Поелементний вивід списку (масиву):

```
for i in range (N):
   print (L [i], end = "")
```

Створення списку. Перший спосіб.

Нехай дано два числа: кількість рядків матриці - **n** і кількість стовпців матриці -**m**.

```
n=3
m=3
A=[0]*n
for i in range(n):
    A[i]=[0]*m
print('A:',A)
```

Відповідь

A: [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

Другий спосіб створення матриці.

Створити порожній список, потім n раз додати в нього новий елемент, який є списком-рядком за допомогою **list.append**(x) який додає елемент у кінець списку.

```
n=3
m=4
A=[]
for i in range(n):
    A.append([0]*m)
print('A:',A)
```

Відповідь

A: [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

Для обробки і виведення списку як правило використовується два вкладених циклу. Перший цикл по номеру рядка, другий цикл по елементах всередині рядка. Наприклад, вивести двовимірний числовий список на екран порядково, розділяючи числа пробілами всередині одного рядка, можна так:

```
n=3
m=3
A=[]
for i in range(n):
    B=[]
```

```
    for i in range(m):
        B.append(int(input()))
    A.append(B)
```

Виводимо матрицю.

```
for i in range(n):
    for j in range(m):
        print(A[i] [j],end='  ')
    print()
```

Відповідь:

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

Те ж саме, але цикли не по індексу, а за значеннями списку:

```
A=[[1,2,3,4],[5,6,7,8],[9,10,11,12]]
for row in A:
    for elem in row:
        print(elem,end='  ')
    print()
```

Відповідь

```
1  2  3  4
```

5 6 7 8

9 10 11 12

Для виведення матриці можливо скористатися методом **join**.

```
for row in A:
    print (' ' .join (list (map (str, row))))
```

```
A=[[1,2],[5,6],[9,10]]
for row in A:
   print(' '.join(list(map(str,row))))
```

Відповідь

1 2

5 6

9 10

Ключові питання

1. Результат роботи програми:

```
a = [[1,2,3],[4,5,6],[7,8,9],[2,3,4],[4,6,8]]
for row in a:
    row[0], row[2] = row[2], row[0]
for row in a:
    for elem in row:
        print(elem, end=' ')
    print()
```

А. Змінюється місцями 0 та 2 строки матриці a

Б. Заповнюється діагональ матриці нулями

В. Змінюється місцями 0 и 2 стовбець матриці a

2. Результат роботи програми:

```
a = [[1,2,3],[4,5,6],[7,8,9],[2,3,4],[4,6,8]]
a[0], a[1] = a[1], a[0]
```

```
for row in a:
    for elem in row:
        print(elem, end=' ')
    print()
```

А. Змінюється місцями 0 и 1 строки матриці a

Б. Змінюється місцями 0 и 2 строки матриці a

В. Змінюється місцями 0 и 2 стовбець матриці a

Завдання на самостійну роботу

1. Створіть матрицю з одних нулів, де кількість рядків матриці - **n** і кількість стовпців матриці - **m**. Де m = №+2, n = №+3, № - остання цифра у списку групи.
2. Написати програму, яка виводить на екран запит на введення елементів матриці розміром **m** та **n**, а потім виводить матрицю на екран. Де m =№+2 та n = №+3. №-остання цифра у списку групи.
3. Дано список A = [[1,3,4,5,6,7,№],[8,9,10,11,№,29,30],[№,2,4,7,8,3,5]]. Де № - остання цифра у списку групи. Виведіть матрицю на екран. Ввести три пробілу між елементами матриці. Зробіть теж саме за допомогою метода join.
4. Дано список A = [[1,3,4,5,6,7,№],[8,9,10,11,№,29,30],[№,2,4,7,8,3,5]]. Де № - остання цифра у списку групи. Написати програму, яка розрахує суми усіх чисел у списку по індексу.
5. Дано список A = [[1,3,4,5,6,7,№],[8,9,10,11,№,29,30],[№,2,4,7,8,3,5]]. Де № - остання цифра у списку групи. Написати програму, яка розрахує суми усіх чисел у списку по значенням строк.
6. Дано список A = [[1,3,4,5,6,7,№],[8,9,10,11,№,29,30],[№,2,4,7,8,3,5]]. Де № - остання цифра у списку групи. Написати програму, яка зводить усі елементи матриці у квадрат.

7. Дано список A = [[1,3,4,5,6,7,№],[8,9,10,11,№,29,30],[№,2,4,7,8,3,5]]. Де № - остання цифра у списку групи. Написати програму, яка зводить усі елементи матриці у квадрат які менше за 5.
8. Дано список A = [[1,3,4,5,6,7,№],[8,9,10,11,№,29,30],[№,2,4,7,8,3,5]]. Де № - остання цифра у списку групи. Написати програму, яка буде находити найбільше число матриці.
9. Серед елементів з непарними номерами знайдіть найбільший елемент масиву, який ділиться на 3.
10. Створити та вивести на екран матрицю 10x10, яка заповнена випадковими числами від 0 до 9.
11. Створити та вивести на екран матрицю 10x10, яка заповнена одиницями, а діагональ нулями.

9. ОБРОБКА ТА ВИВІД ВКЛАДЕНИХ СПИСКІВ

Часто в задачах доводиться зберігати прямокутні таблиці з даними. Такі таблиці називаються матрицями або двовимірними масивами. У мові програмування Пітон таблицю можна представити у вигляді списку рядків, кожен елемент якого є в свою чергу списком, наприклад, чисел. Наприклад, створити числову таблицю з двох рядків і трьох стовпців можна так:

A = [[1, 2, 3], [4, 5, 6]]

Тут перша строчка списку A[0] є списком [1, 2, 3].

A[0][0]= 1,

A[0][1]= 2,

A[0][2]= 3,

A[1][0]=4,

A[1][1]=5,

A[1][2]=6.

Програма використовує два вкладених циклу для розрахунку суми усіх чисел у списку по індексу.

A=[[1,2,3],[5,6,4],[9,10,5]]

```
S=0
for i in range(len(A)):
    for j in range(len(A[i])):
        S+=A[i][j]
print('Сума елементів матриці S= ',S)
```

Відповідь:

Сума елементів матриці S= 45

Програма використовує два вкладених цикла для розрахунку суми усіх чисел у списку по значенням строк.

```
A=[[1,2,3,5],[5,6,4,6]]
S=0
for row in A:
    for elem in row:
        S+=elem
print('Сума елементів матриці S= ',S)
```

Відповідь:

Сума елементів матриці S= 32

Нехай дана квадратна матриця з **n** рядків і **n** стовпців. Необхідно елементам, що знаходяться на головній діагоналі, що проходить з лівого верхнього кута в правий нижній (тобто тих елементів A [i] [j], для яких i == j) присвоїти значення 1, елементам, що знаходяться вище головної діагоналі - значення 0, елементів, що знаходяться нижче головної діагоналі - значення 2. тобто отримати такий масив (приклад для n == 3):

1 0 0

2 1 0

2 2 1

Перший спосіб

Елементи, які лежать вище головної діагоналі - це елементи A [i] [j], для яких i <j, а для елементів нижче головної діагоналі i> j. Таким чином, ми

можемо порівнювати значення i та j . Визначати значення A [i] [j]. Отримуємо наступний алгоритм:

```
for i in range(n):
    for j in range(n):
        if i < j:
            A[i][j] = 0
        elif i > j:
            A[i][j] = 2
        else:
            A[i][j] = 1
```

Наприклад:

```
n=3
m=3
A=[[1,2,3],[2,3,4],[4,5,6]]
for i in range(n):
    for j in range(m):
        if i<j:
            A[i][j]=0
        elif i>j:
            A[i][j]=2
        else:
            A[i][j]=1
for i in range(n):
    for j in range (n):
        print(A[i][j], end=' ')
    print()
```

Відповідь:

```
1 0 0
2 1 0
```

2 2 1

Другий спосіб

Попередній алгоритм виконує одну або дві інструкції if для обробки кожного елемента. Ускладнимо алгоритм та обійдемось без умовних інструкцій.

1. Спочатку заповнимо головну діагональ, для чого нам знадобиться один цикл:

```
for i in range(n):
    A[i][i] = 1
```

2. Заповнимо значенням 0 всі елементи вище головної діагоналі, для чого нам знадобиться в кожній з рядків з номером **i** привласнити значення елементів A [i] [j] для j = i + 1, ..., n-1. Тут нам знадобляться вкладені цикли:

```
for i in range(n):
    for j in range(i + 1, n):
        A[i][j] = 0
```

3. Аналогічно присвоюємо значення 2 елементам A [i] [j] для j = 0, ..., i-1:

```
for i in range(n):
    for j in range(0, i):
        A[i][j] = 2
```

4. Можна також зовнішні цикли об'єднати в один і отримати ще одне, більш компактне рішення:

```
for i in range(n):
    for j in range(0, i):
        A[i][j] = 2
    A[i][i] = 1
    for j in range(i + 1, n):
        A[i][j] = 0
```

Наприклад:

n=3

```
m=3
A=[[1,2,3],[5,6,4],[9,10,5]]
for i in range(n):
    for j in range(0,i):
        A[i][j]=2
    A[i][i]=0
    for j in range (i+1,n):
        A[i][j]=1
for i in range (n):
    for j in range(n):
        print(A[i][j], end=' ')
    print()
```

Відповідь:

```
0 1 1
2 0 1
2 2 0
```

Третій спосіб

Використовується операція повторення списків для побудови чергового рядка списку. і-й рядок списку складається з **i** чисел 2, потім йде одне число 1, потім йде n-i-1 число 0:

```
for i in range(n):
    A[i] = [2] * i + [1] + [0] * (n - i - 1)
```

Наприклад:

```
n=3
m=3
A=[[1,2,3],[5,6,4],[9,10,9]]
for i in range(n):
    A[i]=[2]*i+[0]+[1]*(n-i-1)
for i in range(n):
```

```
    for j in range (n):
        print(A[i][j], end=' ')
    print()
```

Відповідь:

```
0 1 1
2 0 1
2 2 0
```

Наприклад:

Знайти максимальне значення між елементами третього стовпчика.

```
n=3
m=3
A=[[1,2,3],[5,6,4],[9,10,9]]
for i in range(n):
    for j in range(n):
        print(A[i][j], end=' ')
    print()

maximum=A[0][2]
for j in range (n):
    for j in range(n):
        if maximum<A[i][2]:
            maximum=A[i][2]
print('max   ', maximum)
```

Відповідь:

```
1 2 3
5 6 4
9 10 9
max    9
```

Наприклад:

Знайти максимальне значення між елементами другої строки.

```
n=3
m=3
A=[[1,2,3],[5,6,4],[9,10,9]]
for i in range(n):
    for j in range(n):
        print(A[i][j], end=' ')
    print()

maximum=A[1][0]
for j in range (n):
    for j in range(n):
        if maximum<A[1][j]:
            maximum=A[1][j]
print('max   ', maximum)
```

Відповідь:

```
1 2 3
5 6 4
9 10 9
max    6
```

9.1. Підсумовування за допомогою рекурсії.

Рекурсія дозволяє програмами обходити структури, які мають довільні і непередбачувані форми та глибини, наприклад, при плануванні маршрутів в подорож, аналізі мови і проходження по посиланнях в веб-мережі. Рекурсія навіть є альтернативою нескладним циклам і ітераціям, хоча не обов'язково більш простою або ефективною.

Щоб отримати суму списку (або іншій послідовності) чисел, ми можемо або використовувати вбудовану функцію **sum**, або написати власну більш

спеціалізовану версію. Ось як може виглядати спеціальна функція підсумовування, в якій застосовується рекурсія:

```
def mysum(L):
    if not L:
        return 0
    else:
        return L[0]+mysum(L[1:])

>>> mysum([1,2,3,4,5])
15
```

На кожному рівні функція **mysum** рекурсивно викликає саму себе, щоб обчислити суму залишку списку, яка пізніше додається до елементу в голові списку.

Коли список стає порожнім, рекурсивний цикл закінчується і повертається нуль. У разі використання рекурсії такого роду кожен відкритий рівень виклику функції має власну копію локальної області видимості функції в стек викликів часу виконання - тут це означає, що змінна L на кожному рівні різна. Додамо в функцію вивід L на екран і запустимо програму знову, щоб відстежити поточний список на кожному рівні виклику.

```
def mysum(L):
    print(L)
    if not L:
        return 0
    else:
        return L[0]+mysum(L[1:])

>>> mysum([1,2,3,4,5])
Відповідь:
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
```

[3, 4, 5]

[4, 5]

[5]

[]

15

Як легко помітити, підсумовуючий список на кожному рівні рекурсії стає все менше, поки остаточно не спорожніє - кінець рекурсивного циклу. Сума обчислюється при розкручуванні рекурсивних викликів по поверненню.

Рекурсія може бути прямою, як було показано в прикладах чи непрямою, як в наступному прикладі (функція, що викликає іншу функцію, яка знову викликає першу функцію). Сукупний ефект виявляється таким же, хоча на кожному рівні є два виклики функцій замість одного:

```
def mysum(L):
    print(L)
    if not L:
        return 0
    else:
        return nonempty(L)


def nonempty(L):
        return L[0]+mysum(L[1:])
>>> mysum([1.1,2.2,3.3,4.4,5.5])
```

Відповідь:

[1.1, 2.2, 3.3, 4.4, 5.5]

[2.2, 3.3, 4.4, 5.5]

[3.3, 4.4, 5.5]

[4.4, 5.5]

[5.5]

[]

16.5

9.2. Обробка довільних структур.

Рекурсія (або еквівалентні явні алгоритми, засновані на стеку) може вимагатися для обходу структур довільної форми. Як простий приклад рекурсії в такому контексті візьмемо задачу обчислення суми всіх чисел в структурі з вкладеними підсписками такого вигляду:[1, [2, [3, 4], 5], 6, [7, 8]].

Прості оператори циклів не підходять, оскільки це не лінійна ітерація. Вкладених операторів циклу теж не буде достатньо, тому що підсписки можуть бути вкладеними на довільну глибину і в довільній формі. Нема ніякого способу дізнатися, скільки вкладених циклів необхідно написати для обробки всіх випадків. Натомість наступний код пристосовується до такого універсального вкладенню за рахунок застосування рекурсії для відвідування підсписків:

```
def sumtree(L):
    tot=0
    for x in L:
        if not isinstance(x,list):
            tot+=x
        else:
            tot+=sumtree(x)
    return tot

L=[1,[2,[3,4],5],6,[7,8]]
print(sumtree(L))
```

Відповідь: 36

Ключові питання

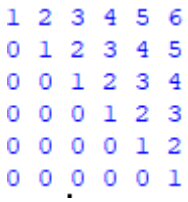
1. Як можливо ввести матрицю за допомогою двох вкладених списків?
2. Описати алгоритми виведення матриці.
3. Описати методику обробки довільних структур даних.

4. Що таке рекурсія?

Завдання на самостійну роботу

1. Дано список A = [[1,7,№],[8,№,29],[№,2,4]]. Де № - остання цифра у списку групи. Виведіть матрицю на екран. Ввести три пробілу між елементами матриці.
2. Присвоїти всім елементам матриці A значення 9. Вивести на екран.
3. Присвоїти елементам, що знаходяться на головній діагоналі, що проходить з лівого верхнього кута в правий нижній (тобто тих елементів A [i] [j], для яких i == j) значення 3, елементам, що знаходяться вище головної діагоналі - значення 4, елементів, що знаходяться нижче головної діагоналі - значення 5. Програму написати трьома способами.
4. Дано список A = [[1,7,№],[8,№,29],[№,2,4]]. Де № - остання цифра у списку групи. Введіть список з екрану. Знайти мінімальне значення між елементами другого стовпчика. Вивести отримані значення.
5. Дано список A = [[1,7,№],[8,№,29],[№,2,4]]. Де № - остання цифра у списку групи. Введіть список з екрану. Знайти мінімальне значення між елементами третьої строки. Вивести отримані значення.
6. Дано - двовимірний масив розміром m n (будь який). Сформувати новий масив замінивши позитивні елементи одиницями, а негативні нулями. Вивести обидва масиви.
7. Матриця 5x5 розташована у одномірному масиві по рядкам. Змінити місцями другий та третій рядок. Результат представити у вигляді матриці.
8. Матриця 5x5 розташована у одномірному масиві по рядкам. Змінити місцями другий та третій стовпчик. Результат представити у вигляді матриці.
9. У матриці 6x6 замінить третій рядок й другий стовпчик нулями, крім елемента розташованого на їх перехресті.

10. Дана дійсна матриця 10x10. Знайти суму найбільших значень елементів її рядків.
11. Дана цілочисельна квадратна матриця 6x6. Заповнить її наступним чином.



10. АНОНІМНІ ФУНКЦІЇ: ВИРАЖЕННЯ LAMBDA

Крім оператора **def** в **Python** також пропонується форма вираження, яка генерує об'єкти функцій. Через схожість з інструментом в мові **Lisp** вона називається **lambda**. Подібно **def** такий вислів створює функцію, яка буде викликатися пізніше, але повертає сам об'єкт функції, не привласнюючи його імені. З цієї причини вираження **lambda** іноді називають **анонімними** (тобто безіменними) функціями.

На практиці вони часто застосовуються для того, щоб вбудувати визначення функції в рядок або відкласти виконання частини коду.

Загальна форма **lambda** виглядає як ключове слово **lambda**, за яким слідують один або більше аргументів (дуже схоже на список аргументів, укладений в круглі дужки в заголовку **def**) і далі вираз після двокрапки:

lambda аргумент 1, аргумент2. . . аргумент№: вираз, що використовує аргументи

f=lambda x,y,z: x+y+z

Відповідь: 9

Імені **f** присвоюється об'єкт функції, створений виразом **lambda**; так працює оператор **def**, але він виробляє присвоювання автоматично. Як і в разі **def**, для аргументів **lambda** можна вказувати стандартні значення:

f=lambda x,y,z: x+y+z

>>> f('111','222','333')

Відповідь: '111222333'

Функції **lambda**, на відміну від звичайної, не потрібна інструкція **return**, а в іншому, поводиться ідентично. Вираз **lambda** корисний як свого роду коротке умовне позначення функції, яке дозволяє вбудовувати визначення функції всередину коду, де воно застосовується. Вираз **lambda** зовсім необов'язково використовувати завжди. Замість нього використовується оператор **def**, якщо функція вимагає потужності повних операторів, яку **lambda** не здатна легко надати. Але в сценаріях, де потрібно всього лише вбудовувати невеликі частини виконуваного коду в місцях їх застосування, вираження **lambda** виявляться більш простими кодовими конструкціями.

Вирази **lambda** також широко використовуються при реалізації таблиць переходів, які представляють собою списки або словники дій, що підлягають виконанню за запитом.

```
L=[lambda x: x**2,
   lambda x: x**3,
   lambda x: x**4,]
for d in L:
   print(d(3))
print(L[0] (4))
```

Відповідь:

9

27

81

16

10.1. Інструменти функціонального програмування

Вбудована функція **map.** Одним з найбільш частіших дій, які виконуються в програмах зі списками і іншими послідовностями, є застосування якоїсь операції до кожного елементу і накопичення результатів - вибір стовпців в таблицях бази даних, збільшення значень в полях із

заробітною платою співробітників компанії, розбір вкладень в повідомленнях електронної пошти і т.д.

Вбудована функція **map** застосовує передану функцію до кожного елементу в ітеруемому об'єкті і повертає список, що містить всі результати викликів функції.

```
>>> counters=[1,2,3,4]
>>> def inc(x): return x+10

>>> list(map(inc,counters))
```

Відповідь:

[11, 12, 13, 14]

def inc(x): return (x + 10) – Функція яка буде виконуватись.

list(map(inc,counters))- Накопичення результатів.

У списку **map** - викликає **inc** на кожному елементі списку і збирає все повернені значення в новий список. Для виведення всіх результатів використовується виклик **list**.

Через те, що таке використання куля еквівалентно циклам **for**, додавши трохи коду, ви можете отримати універсальну утиліту відображення:

```
>>> def inc(x): return x++10

>>> def mymap(func, seq):
        res=[]
        for x in seq: res.append(func(x))
        return res

>>> list(map(inc,[1,2,3]))
```

Відповідь: [11, 12, 13]

Функцію **map** можна використовувати більш розвиненими способами, ніж показано. Отримавши в якості аргументів кілька послідовностей, **map**

передає витягнуті з послідовностей елементи як індивідуальні аргументи функції **pow**:

>>> pow(3,4)

Відповідь: 81

>>> list(map(pow,[1,2,3],[2,3,4]))

Відповідь: [1, 8, 81]

10.2. Вибір елементів з ітерованих об'єктів: filter

Функція **map** - початковий і щодо прямолінійний представник інструментального набору для функціонального програмування в **Python**. Її близькі родичі, **filter** та **reduce**, вибирають елементи з ітеруемих об'єктів на основі перевірочної функції і застосовують функції до пар елементів відповідно.

Через повернення ітеруемого об'єкта функція **filter** (подібно **range**) вимагає виклику **list** при відображенні всіх її результатів. Наприклад **filter** вибирає з послідовності елементи більше нуля:

>>> list(range(-5,5))

Відповідь: [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(filter((lambda x: x>0), range(-5, 5)))

Відповідь: [1, 2, 3, 4]

Як і **map**, функція **filter** приблизно еквівалентна циклу **for**, але вона є вбудованої, лаконічною і часто більш швидкою:

```
>>> res=[]
>>> for x in range(-5, 5):
        if x>0:
                res.append(x)

>>> res
```

Відповідь: [1, 2, 3, 4]

Також подібно **map** функцію **filter** можна емалювати за допомогою синтаксису спискового включення з часто більш простими результатами і за допомогою схожого генераторного виразу, коли бажано відкладати випуск результатів.

>>> [x for x in range(-5, 5) if x>0]

Відповідь: [1, 2, 3, 4]

10.3. Комбінування елементів з ітерованих об'єктів: reduce.

Виклик функції **reduce**, яка знаходиться в модулі **functools**, виглядає складніше. Вона приймає ітеруемий об'єкт, що підлягає обробці, але сама не є ітеруемим об'єктом, а повертає одиночний результат. Нижче показані два виклики **reduce**, які обчислюють суму і добуток елементів у списку:

>>> from functools import reduce

>>> reduce((lambda x,y:x*y),[1,2,3,4])

Відповідь: 24

>>> reduce((lambda x,y:x+y),[1,2,3,4])

Відповідь: 10

На кожному кроці **reduce** передає поточну суму або добуток разом з черговим елементом зі списку зазначеної функції **lambda**. За замовчуванням початкове значення відповідає першому елементу послідовності.

Інструменти **map** і **filter** - головні члени раннього інструментального набору для функціонального програмування на **Python**. Вони відображають операції на ітеровані об'єкти і накопичують результати. Через те, що ця задача настільки поширена при написанні з'явився новий вираз - спискові включення, які володіють навіть більшою гнучкістю, ніж інші інструменти.

10.4. Формальний синтаксис включень

Насправді спискові включення навіть більш універсальні. Їх найпростіша форма передбачає зазначення виразу, який накопичується і одиночної конструкції **for**:

[*Вираз for мета in ітеруємий_об'єкт*]

Незважаючи на необов'язковість всіх інших частин, вони дозволяють висловлювати розвиненіші ітерації - в списковому включенні допускається записувати будь-яку кількість вкладених циклів **for**, кожний з яких може мати необов'язкову асоційовану перевірку **if**, що діє як фільтр. Загальна структура спискових включень виглядає наступним чином:

[Вираз for мета1 in ітеруємий_об'єкт1 if умова1
for мета2 in ітеруємий_об'єкт2 if умова 2. . .
for мета № in ітеруємий_об'єкт№ if умова №]

Такий самий синтаксис успадкований включеннями множин і словників, а також генераторними виразами, які з'явилися пізніше, хоча вони використовують інші символи (фігурні дужки або часто необов'язкові круглі дужки), а включення словника починається з двох виразів, розділених двокрапкою (для ключа і значення).

Наприклад:

>>> [x+y for x in 'spam' for y in 'SPAM']

Відповідь ['sS', 'sP', 'sA', 'sM', 'pS', 'pP', 'pA', 'pM', 'aS', 'aP', 'aA', 'aM', 'mS', 'mP', 'mA', 'mM']

Наприклад:

>>> res=[x+y for x in[0,1,2] for y in[100,200,300]]

>>> res

Відповідь: [100, 200, 300, 101, 201, 301, 102, 202, 302]

Кожна конструкція **for** може мати асоційований фільтр if незалежно від того, наскільки глибоко вкладені цикли.

Наприклад:

>>> [x+y for x in 'spam' if x in 'sm' for y in 'SPAM'if y in ('P','A')]

Відповідь: ['sP', 'sA', 'mP', 'mA']

Наприклад:

```
>>> [x+y+z for x in 'spam' if x in 'sm'
        for y in 'SPAM'if y in ('P','A')
    for z in'246' if z>'2']
```

Відповідь: ['sP4', 'sP6', 'sA4', 'sA6', 'mP4', 'mP6', 'mA4', 'mA6']

Вираз комбінує парні числа від 0 до 4 з непарними числами від 0 до 4.

Конструкції **if** фільтрують елементи на кожній ітерації.

Наприклад:

```
>>> [(x,y) for x in range(5) if x%2==0
        for y in range(5) if y%2==1]
```

Відповідь [(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]

10.5. Спискові включення і матриці

Наприклад, в наступному коді визначаються дві матриці 3 х 3 у вигляді списків, що складаються з вкладених списків:

```
>>> M=[[1,2,3],
      [4,5,6],
      [7,8,9]]
>>> N=[[2,2,2],
      [3,3,3],
      [4,4,4]]
>>> M[1]
```

Відповідь: [4, 5, 6]

```
>>> M[1][2]
```

Відповідь: 6

Спискові включення є потужним інструментом для обробки

таких структур як матриці, тому що вони автоматично переглядають рядки і стовпці.

```
>>> M=[[1,2,3],
      [4,5,6],
      [7,8,9]]
>>> N=[[2,2,2],
      [3,3,3],
      [4,4,4]]
>>> [row[1] for row in M]
```

Відповідь: [2, 5, 8]

```
>>> [M[row][1] for row in (0,1,2)]
```

Відповідь: [2, 5, 8]

```
>>> [M[row][0] for row in (0,1,2)]
```

Відповідь [1, 4, 7]

За заданими позиціями ми також можемо виконувати завдання на зразок вилучення діагоналі. У першому з наведених далі виразів із застосуванням функції **range** генерується список зсувів і проводиться індексація з однаковими номерами рядків і стовпців, що вибирає M [0] [0], потім M [1] [1] і т.д. У другому рядку індекс стовпця врівноважується для вилучення M [0] [2], M [1] [1] і т.д. Ми припускаємо, що матриця має рівну кількість рядків і стовпців.

Наприклад:

```
>>> M=[[1,2,3],
      [4,5,6],
      [7,8,9]]
>>> [M[i][i] for i in range(len(M))]
```

Відповідь: [1, 5, 9]

Витягування зворотної діагоналі:

```
>>> M=[[1,2,3],
```

```
    [4,5,6],
    [7,8,9]]
>>> [M[i][len(M)-1-i] for i in range(len(M))]
```

Відповідь: [3, 5, 7]

Приклад зміни кожного члена матриці на 10:

```
>>> M=[[1,2,3],
    [4,5,6],
    [7,8,9]]
>>> [[col+10 for col in row] for row in M]
```

Відповідь [[11, 12, 13], [14, 15, 16], [17, 18, 19]]

Приклад перемноження матриць:

```
>>> M=[[1,2,3],
    [4,5,6],
    [7,8,9]]
>>> N=[[2,2,2],
    [3,3,3],
    [4,4,4]]
>>> [[M[row][col]*N[row][col]for col in range(3)]for row in range(3)]
```

Відповідь: [[2, 4, 6], [12, 15, 18], [28, 32, 36]]

Ключові питання

1.Для чого використовують анонімну функцію **lambda**?

2. Для чого використовують вбудовану функцію **map**?

3. Для чого використовують вбудовану функцію **filter**?

4. Для чого використовують вбудовану функцію **reduce**?

Завдання на самостійну роботу

1. Написати програму, яка використовує анонімну функцію **lambda**.
2. Написати програму, яка використовує вбудовану функцію **map**.
3. Написати програму, яка використовує вбудовану функцію **filter**.
4. Написати програму, яка використовує вбудовану функцію **reduce**.
5. Написати програму, яка комбінує непарні числа від 0 до 7 з парними числами від 0 до 7 за допомогою списків включень.

11. ГЕНЕРАТОРНІ ФУНКЦІЇ І ВИРАЗИ

Генераторні функції записуються як нормальні оператори **def**, але в них застосовуються оператори **yield**, щоб повертати по одному результату за раз, призупиняти виконання зі збереженням стану і відновлювати його між видачами.

В великих програмах генератори можуть бути краще в плані пам'яті і продуктивності. Вони дозволяють функціям уникнути виконання всієї роботи заздалегідь, що особливо корисно, коли результуючі списки великі або отримання кожного значення вимагає тривалих обчислень. Генератори розподіляють час, необхідний для виробництва серії значень, за всіма ітераціям циклу. Крім того, для ускладнених сценаріїв застосування генераторів здатне запропонувати більш просту альтернативу ручному збереженню стану між ітераціями. В об'єктах класів - генератори забезпечують автоматичне збереження і відновлення змінних, доступних в області видимості функцій.

11.1. Генераторні функції yield або return

Генераторні функції тісно пов'язані з поняттям протоколу ітерації в **Python**. Об'єкти ітераторів визначають метод, який або повертає черговий елемент в ітерації, або ініціює спеціальне виключення **Stopiteration** для закінчення ітерації. Ітератор ітеруємого об'єкта спочатку витягується за допомогою вбудованої функції **iter**, хоча цей крок нічого не робить для об'єктів, які самі є ітераторами.

Цикли **for** в **Python** і всі інші ітераційні контексти використовують протокол ітерації для проходу по генератору послідовностей або значень, якщо протокол підтримується (якщо ні, тоді ітерація натомість робить багаторазову індексацію послідовностей). Будь-який об'єкт, що підтримує такий інтерфейс, працює з усіма ітераційним інструментами.

Для підтримки протоколу ітерації функції, що містять оператор yield, компілюються в особливий спосіб як генератори - вони не будуть нормальними функціями, а будуються з метою повернення об'єкта з очікуваними методами з протоколу ітерації. При наступному виклику вони повертають об'єкт генератора, який підтримує інтерфейс ітерації з автоматично створеним методом по імені, призначеним для запуску або відновлення виконання.

Генераторні функції можуть також мати оператор **return**, який поряд з переміщенням за кінець блоку **def** просто припиняє генерацію значень - формально за рахунок ініціювання виключення **Stopiteration** після всіх звичайних дій по виходу з функції. З точки зору викликаючого коду, метод генератора відновлює виконання функції до тих пір, поки або не повернеться наступний результат **yield**, або до виникнення виключення **StopIteration**.

Сукупний ефект в тому, що генераторні функції, які записані як оператори **def**, що містять оператори **yield**, автоматично робляться доступними для протоколу ітерації і тому можуть застосовуватися в будь-якому ітераційному контексті, щоб виробляти результати з плином часу і за запитом.

У наступному коді визначається генераторная функція, яку можна застосовувати для генерації квадратів серії чисел з плином часу:

```
>>> def gensquares(N):
        for i in range(N):
            yield i**2

>>> for i in gensquares(5):
        print(i,end=' : ')
```

Відповідь: 0 : 1 : 4 : 9 : 16 :

Функція **gensquares** видає значення і тому повертає управління коду який викликається на кожній ітерації циклу; при поновленні її виконання відновлюється попередній стан, включаючи останні значення змінних і N, а управління знову підхоплюється безпосередньо після оператора **yield**. Скажімо, коли gensquares використовується в тілі циклу **for**, перша ітерація починає функцію і отримує перший результат; потім на кожній ітерації циклу управління повертається функції після оператора **yield**.

Щоб закінчити генерацію значень, функції або застосовують оператор **return** без значення, або дозволяють потоку управління дійти до кінця тіла функції.

11.2 Генераторні вирази.

Ітеруєми об'єкти які зустрічаються з включеннями. Генераторні вирази схожі на спискові включення, але вони не будують результуючий список, а повертають об'єкти, які виробляють результати за запитом. Синтаксично генераторні вирази схожі на нормальні спискові включення і підтримують весь їх синтаксис, в тому числі фільтри **if** і вкладення циклів, але вони розміщаються в круглі дужки, а не в квадратні (подібно кортежам круглі дужки часто необов'язкові).

У точності як генераторні функції, генераторні вирази забезпечують оптимізацію витрат пам'яті - вони не вимагають створення відразу всього результуючого списку, що відбувається в разі спискового включення в квадратних дужках. Також подібно генераторним функцій вони поділяють роботу з випуску результатів на невеликі тимчасові інтервали - результати видаються поступово замість того, щоб викликати код та очікувати створення повного набору в єдиному виклику.

З іншого боку, на практиці генераторні вирази можуть виконуватися трохи повільніше спискових включень, а тому їх найкраще застосовувати для дуже великих результуючих наборів або в додатках, які не можуть чекати генерації повних результатів.

Наприклад:

Генераторний вираз будує інтеруємий об’єкт.

```
>>> (x**2 for x in range(4))
<generator object <genexpr> at 0x042415B0>

>>> list(x**2 for x in range(4))
```

Відповідь: [0, 1, 4, 9]

Для примусового формування всіх результатів відразу використовуємо вбудовану функцію **list**.

Ключові питання

1. Для чого використовують генераторну функцію **yield**?
2. Для чого використовують генераторну функцію **return**?
3. Що таке генераторний вираз?

Завдання на самостійну роботу

1. Написати програму, яка використовує генераторну функцію **yield**.
2. Написати програму, яка використовує генераторну функцію **return**.
3. Написати програму, яка використовує генераторний вираз.

12. ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

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.

12.1. Створення класів і об'єктів.

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 B:
...     n = 5
...     def adder(v):
...             return v + B.n
...
>>> B.n
```

Відповідь: 5

```
>>> B.adder(4)
```

Відповідь: 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

```
>>> 1 = B()
>>> 1.n
```

Відповідь 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 класу:

>>> 1.n = 10

>>> 1.n

Відповідь: 10

>>> B.n

Відповідь: 5

Тут змінні 1.n і B.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.

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)

перетворюється у вираз

B.adder(1, 100)

Таким чином, інтерпретатор спробував передати у метод adder() класу B два фрагмента - 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'ктом, 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)
```

Відповідь: 13

```
>>> m.adder(4)
```

Відповідь: 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
```

Відповідь: True

```
>>> l.n is B.n
```

Відповідь: 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.

Прийнято привласнювати полях, 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()
```

Відповідь: 'Bob'

```
>>> second.getName()
```

Відповідь: No name

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

12.1.1. 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
```

Тоді створення об'єкта можливе без полів. Для встановлення імені метод set_name() неообхідно 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)
```

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())
```

12.1.2. Деструктор класу.

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".

Ключові питання

1.Що таке об'єктно-орієнтованого програмування?

2. Що таке клас ?

3. Як виглядає конструктор класу?

4. Як виглядає деструктор класу?

Завдання на самостійну роботу

1. Написати програму, яка використовує конструктор класу.
2. Написати програму, яка використовує деструктор класу.
3. 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.
4. У класу **Person** є метод, який повертає рядок, що включає в себе всю інформацію про студента.
5. Клас **Person** містить деструктор, який виводить на екран фразу "Ви отримали стипендію ..." (замість три крапки повинні виводитися ім'я та прізвище об'єкта).
6. В основній гілці програми створіть три об'єкти класу **Person**. Подивіться інформацію про студента і об’явить, що він отримав стипендію.
7. 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.

13. УСПАДКУВАННЯ

Успадкування (наслідування, inheritance) - механізм утворення нових класів на основі використання властивостей і функціоналу вже існуючих класів.

Ключовими поняттями наслідування є підклас (**subclass**) і суперклас (**super class**). Суперклас ще називають базовим (**base class**) або батьківським (**parent class**), а підклас - похідним (**derived class**) або дочірнім (**child class**).

Підклас успадковує властивості, методи та інші публічні атрибути з базового класу. Він також може перевизначати (**override**) методи базового класу. Якщо підклас не визначає свій конструктор, він успадковує конструктор базового класу за замовчуванням.

В **Python** синтаксис для наслідування класів виглядає наступним чином:

```
class <subclass>(<superclass>):
    <subclass attributes>
```

Розглянемо на прикладі.

```
>>> class Base:
...     def __init__(self):
...             self.base_prop = 'base property'
...     def method(self):
...             print("Це метод з класу Base.")
...             print("У об'єкта класу Base є атрибут base_prop, його значення:", self.base_prop)
...
>>> class Child(Base):
...     def child_method(self):
...             print("Це метод з класу Child.")
...             print("Об'єкт класа Child має атрибут base_prop. Його створено в успадкованому конструкторі класу Base:", self.base_prop)
...
>>> c = Child()
>>> c.method()
Це метод з класу Base.
У об'єкта класу Base є атрибут base_prop, його значення: base property
>>> c.child_method()
Це метод з класу Child.
Об'єкт класу Child має атрибут base_prop. Його створено в успадкованому конструкторі класу Base: base property
```

Що відбувається у цьому прикладі?

- Клас **Child** успадковує від класу **Base** два методи: конструктор і метод method().
- Також клас **Child** має свій власний метод: child_method().
- При створенні об'єкта класа **Child** буде викликано успадкований конструктор класу **Base**. У конструкторі для об'єкта створюється атрибут base_prop. У об'єктів класу **Child** теж буде створено цей атрибут.

13.1. Ієрархія успадкування.

Клас може бути успадкованим від класу, який у свою чергу було успадковано від іншого класу. Коли розглядають увесь ланцюжок успадкованих і базових класів, говорять про **ієрархію успадкування**.

В **Python** є вбудований клас який має назву **object**. Від цього класу явно чи неявно успадковуються усі інші класи, як вбудовані, так і ті, що створені. Якщо при створені класу ви не вказуєте базовий клас, то неявним чином ваш клас буде успадковане від **object**. Отже наступні оголошення класу рівносильні:

```
class A:
    pass

class A():
    pass

class A(object):
    pass
```

Розглянемо наступну ієрархію класів:

```
>>> class A: pass
...
>>> class B(A): pass
...
>>> class C(B): pass
```

...

>>> c_obj = C()

>>> c

<__main__.Child object at 0x00000151150290F0>

Дізнатись, чи є певний клас підкласом іншого класу по всій ієрархії успадкування, можна за допомогою вбудованої функції **issubclass()**:

>>> issubclass(C, B)

True

>>> issubclass(C, A)

True

>>> issubclass(C, object)

True

>>> issubclass(B, C)

False

Також можна дізнатись чи є певний об'єкт екземпляром класу враховуючи всю ієрархію успадкування:

>>> isinstance(c_obj, C)

True

>>> isinstance(c_obj, B)

True

>>> isinstance(c_obj, object)

True

>>> isinstance('some striing', C)

False

>>> isinstance('some striing', object)

True

У кожного класу є спеціальний атрибут, в якому міститься базовий клас, точніше кортеж який містить базові класи (чому саме так - трохи далі):

>>> C.__bases__

(<class '__main__.B'>,)

```
>>> B.__bases__
(<class '__main__.A'>,)
>>> A.__bases__
(<class 'object'>,)
```

13.2 Успадкування і приватні атрибути.

Атрибути, які починаються з двох символів підкреслення (але не закінчуються ними) є приватними атрибутами класу. Поза видимістю класу до таких атрибутів застосовується механізм **name mangling** (спотворення імені), тобто такі атрибути "поза класом" будуть мати інші імена (клас+атрибут), у тому числі і в успадкованих класах. Це дозволяє "приховати" внутрішню реалізацію класа навіть для класів, які від нього успадкуються.

Наприклад:

```
>>> class Base:
...     def __init__(self):
...             self.__prop = 'property'
...     def base_method(self):
...             print("Це метод з класа Base.")
...                print("У об'єкта класа Base є атрибут __prop, його значення:", self.__prop)
...
>>> class Child(Base):
...     def child_method(self):
...             print("Це метод з класа Child.")
...              print("Спробуємо дістатись до атрибута з метода успадкованого класа:", self.__prop)
...
>>> c = Child()
>>> c.base_method()
```

```
Це метод з класа Base.
У об'єкта класа Base є атрибут __prop, його значення: property
>>> c.child_method()
Це метод з класа Child.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in child_method
AttributeError: 'Child' object has no attribute '_Child__prop'
```

13.3 Лінеаризація.

Дочірній клас може не мати певного атрибута, але він може успадкувати його від базового класу. Для пошуку атрибутів в ієрархії класів використовується лінеаризація класів.

Лінеаризація - це черговість, при якій проводиться пошук зазначеного атрибута в ієрархії класів. Використовуючи лінеаризацію відбувається пошук атрибутів в ієрархії класів. При простому успадкуванні алгоритм пошуку атрибутів виглядає наступним чином:

- якщо атрибут, до якого відбувається доступ, не знайдено в поточному класі, то виконується його пошук в базовому класі;
- якщо атрибут не знайдено і в базовому класі, то виконується його пошук в базовому класі базового класу;
- пошук відбувається рекурсивно аж до класу **object**;
- якщо атрибут не знайдено і в класі **object**, то отримуємо виняткову ситуацію.

Наприклад:

```
>>> class A:
...     def f1(self):
...             print('f1 method in class A')
...
>>>
```

```
... class B(A):
...     def f2(self):
...         print('f2 method in class B')
...
>>>
... class C(B):
...     def f2(self):
...         print('f2 method in class C')
...     def f3(self):
...         print('f3 method in class C')
...
>>> obj = C()
>>> obj.f3()
f3 method in class C
>>> obj.f2()
f2 method in class C
>>> obj.f1()
f1 method in class A
>>> obj.f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute 'f'
>>>
```

В **Python** лінеаризація ще називається MRO - Method Resolution Order, порядок вирішення методів. Лінеаризація для певного класу знаходиться в його спеціальному атрибуті __mro__:

```
>>> C.__mro__
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

>>>

Але частіше користуються атрибутом-методом класу, який повертає не кортеж, а одразу список:

>>> C.mro()

[<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]

13.4. Перевизначення і пошук методів.

Уявімо ситуацію, що в базовому класі, від якого ми будемо успадковувати наш новий клас, вже реалізовано певний метод, котрий підходить нам по своїй функціональності, але у ньому не вистачає певних речей, або ж нам треба дещо змінити його функціонал. Звісно, що ми можемо повністю переписати цей метод у нашому новому класі, але з великою ймовірністю ми стикнемось з повторним використання коду. І якщо, припустимо, ми вносимо зміни в метод базового класу, то також з зміни нам доведеться вносити і в аналогічний метод нашого нового класу, що є небажаним (підвищується ймовірність припуститись помилки, зайва робота в решті решт).

Якщо в дочірньому класі певний атрибут був перевизначений, а потрібен доступ до відповідного атрибута базового класу, то **Python** це можливо зробити двома способами. Один з них полягає у тому, що ми явно вказуємо базовий клас, відповідний атрибут і, при необхідності, передаємо екземпляр дочірнього класу (параметр self).

Наприклад:

```
>>> class Person:
...     def __init__(self, name):
...         self.name = name.title()
...     def say_hello(self):
...         print('Hi, I am', self.name)
...
```

```
>>> p = Person('john')
>>> p.say_hello()
Hi, I am John
>>>
```

В конструкторі класу відбувається певна маніпуляція зі вхідними даними. Нам треба створити клас, який описував би не просто людину, а співробітника певної організації. Співробітник має усі атрибути, які має і людина (зокрема ім'я), власне будь-який співробітник і є людиною, тому логічно успадкуватись від класу **Person**. Крім того співробітник має ще й заробітню плату.

```
>>> class Employee(Person):
...     def __init__(self, name, salary):
...         Person.__init__(self, name)
...         self.salary = salary
...     def say_hello(self):
...         Person.say_hello(self)
...         print('My salary is', self.salary)
...
>>> e = Employee('JANE', 120)
>>> e.say_hello()
Hi, I am Jane
My salary is 120
```

В конструкторі дочірнього класу ми викликаємо конструктор базового класу при цьому передаючи йому екземпляр дочірнього класу і необхідні дані для ініціалізації атрибутів. В конструкторі базового класу відбувається певна маніпуляція над вхідними даними і відбувається ініціалізація атрибута **name**. І вже потім в конструкторі дочірнього класу відбувається ініціалізація атрибута **salary**. Аналогічно з метода say_hello() дочірнього класу викликається відповідний метод базового класу.

Недоліки такого підходу:

- ускладнюється підтримка коду якщо нам треба щось поміняти в ієрархії класів.
- логіка коду чітко прив'язана до ієрархії успадкування класів і схильна до помилок, особливо при використанні множинного успадкування.

13.5. Множинне успадкування.

Множинна спадковість - властивість деяких об’єктно-орієнтованих мов програмування, в яких класи можуть успадкувати поведінку і властивості більш ніж від одного суперкласу (безпосереднього батьківського класу). Це відрізняється від простого спадкування, у випадку якого клас може мати тільки один суперклас.

Python підтримує множинну спадковість:

```
>>> class Horse:
...     def run(self):
...             print("Я біжу!")
...
>>>
... class Eagle:
...     def fly(self):
...             print("Я лечу!")
...
>>>
... class Pegasus(Horse, Eagle):
...     pass
...
>>>
... p = Pegasus()
>>> p.run()
Я біжу!
>>> p.fly()
```

Я лечу!

У прикладі клас **Pegasus** успадкував від класу **Horse** метод **run(),** а від класу Eagle - метод **fly()**. Зауважимо, що у множинної спадковості є як переваги, так і недоліки. До переваг можна віднести те, що множинна спадковість дозволяє проектувати доволі складні і гнучкі ієрархії класів. Але разом з тим використання цього механізму може призвести до появи у коді доволі серйозних помилок. Сама наявність множинної спадковості може бути індикатором наявності помилок в проектуванні архітектури вашого застосунку, тому що множинна спадковість на практиці використовується вкрай рідко. І якщо у вас виникає думка використати множинну спадковість, то треба добре подумати, чи правильно ви розбиваєте предметну область на класи взагалі. І тільки якщо ви дійдете до висновку, що використання множинної спадковості доречне і дійсно може спростити код, тоді вже можете використовувати її.

13.6. Лінеарізація і множинна спадковість.

При множинній спадковості виникає питання: а як саме треба виконувати лінеарізацію класів? А також варіантів обійти всю ієрархію класів коли у одного класу є більше ніж один базовий клас можна побудувати декілька. При цьому не повинна "ламатись" логічна структура успадкування, це напряму впливає на алгоритм пошуку атрибутів в ієрархії класів.

Самий простий приклад де виникає неоднозначність пошуку методів - задача ромбовидного успадкування. Вивчіть самостійно наступний код і зробіть припущення, що буде виведено в результаті його виконання:

```
class Animal:
    def hello(self):
        print('Hello, I am Animal')


class Eagle(Animal):
    def hello(self):
        print('Hello, I am Eagle')
```

```
class Horse(Animal):
    pass

class Pegasus(Eagle, Horse):
    pass

p = Pegasus()
p.hello()
```

Підходів побудови лінеаризації при множинній спадковості існує декілька, в різних мовах програмування можуть використовуватись різні підходи. А в декотрих мовах програмування, які підтримують парадигму ООП, множинне успадкування відсутнє.

В **Python** використовується алгоритм C3-лінеаризації, котрий дозволяє побудувати стійкий список з самого класу і усіх його предків (батьків і прабатьків). Цей алгоритм вирішує більшість проблем при лінеаризації множинного успадкування. Алгоритм відносно складний, зовсім спрощено його можна представити так:

- у список додається клас об'єкта (далі - дочірній клас)
- у список додаються базові класи дочірнього класу у тому порядку, як вони вказані при декларації дочірнього класу
- у список додаються базові класи базових класів і так далі рекурсивно до класу **object**
- якщо якийсь клас опиняється у списку двічі - залишається тільки останнє його входження

Як результат, ми рухаємось по рівням, не звертаємось до базового класу до того, як звернемось до усіх його нащадків, навіть якщо нащадків у цього базового класу декілька.

Алгоритм забезпечує пошук перевизначеного метода базового класу, якщо цей метод перевизначено хоча б у одному нащадку цього базового класу.

```
>>> class Animal:
...     pass
...
>>>
>>> class Eagle(Animal):
...     pass
...
>>>
>>> class Horse(Animal):
...     pass
...
>>>
>>> class Pegasus(Eagle, Horse):
...     pass
...
>>> for c in Pegasus.mro():
...     print(c.__name__)
...
Pegasus
Eagle
Horse
Animal
object
>>>
```

Як видно з прикладу, лінеарізація обходить усі класи нашої ієрархії.

Класи Eagle і Horse йдуть у тій послідовності, як перераховані при визначенні класа Pegasus. Давайте поміняємо їх місцями і подивимось як це вплине на лінеарізацію:

```
>>> class Pegasus(Horse, Eagle):
...     pass
```

```
...
>>> for c in Pegasus.mro():
...     print(c.__name__)
...
Pegasus
Horse
Eagle
Animal
object
>>>
```

13.7. Пошук методів і лінеарізація.

Як згадувалось раніше, доступ до атрибутів базового класу можна отримати шляхом прямої вказівки назви класу і, власне, атрибута. Існує ще один спосіб, який позбавлений недоліків попереднього.

Існує спеціальний клас **super**, екземпляри якого є спеціальними проксі-об'єктами (об'єктами-посередниками), які прив'язані до даної ієрархії класів і які надають доступ до атрибутів наступного класу в лінеаризації того класу, в якому було створено об'єкт **super**.

Таким чином, за допомогою **super** можна отримати доступ до атрибутів суперкласу, не вказуючи його імені, причому це буде давати коректні результати навіть при використанні множинного успадкування.

Наприклад:

```
super(MyClass, self).method()
```

Якщо при створенні екземпляра класу **super** не вказувати параметри, то автоматично будуть отримані поточні клас і його екземпляр:

```
super().method()
```

Приклад використання super() при простому успадкуванні:

```
>>> class Person:
```

```
...     def __init__(self, name):
...         self.name = name.title()
...     def say_hello(self):
...         print('Hi, I am', self.name)
...
>>>
... class Employee(Person):
...     def __init__(self, name, salary):
...         super().__init__(name)
...         self.salary = salary
...     def say_hello(self):
...         super().say_hello()
...         print('My salary is', self.salary)
...
>>>
>>> e = Employee('janE', 120)
>>> e.say_hello()
Hi, I am Jane
My salary is 120
>>>
```

З класу **Employee** ми отримуємо доступ до атрибутів класу **Person** за допомогою **super()**. Тепер не треба вказувати навіть поточний екземпляр класу.

Тепер звернемось до множинної спадковості:

```
>>> class Animal:
...     def __init__(self):
...         self.can_run = False
...         self.can_fly = False
...
>>>
>>> class Horse(Animal):
```

```
...     def __init__(self):
...             super().__init__()
...             self.can_run = True
...
>>>
>>> class Eagle(Animal):
...     def __init__(self):
...             super().__init__()
...             self.can_fly = True
...
>>>
>>> class Pegasus(Horse, Eagle):
...     pass
...
>>> p = Pegasus()
>>> p.can_run
True
>>> p.can_fly
True
>>>
```

Щоб краще зрозуміти, як це працює, давайте спочатку вивчимо лінеаризацію класа Pegasus:

```
>>> Pegasus.mro()
[<class '__main__.Pegasus'>, <class '__main__.Horse'>, <class '__main__.Eagle'>, <class '__main__.Animal'>, <class 'object'>]
>>>
```

Тепер покроково:

- В класі **Pegasus** конструктора не оголошено, отже згідно лінеаризації викликаємо конструктор класу Horse.

- В конструкторі **Horse** за допомогою **super()** викликається конструктор "попереднього" класу, згідно MRO це буде конструктор класу **Eagle**.
- В конструкторі **Eagle** за допомогою **super()** викликається конструктор "попереднього" класу, згідно MRO це буде конструктор класу **Animal**.

Ключові питання

1. Що називається успадкуванням в ООП?
2. Що називають ієрархію успадкування?
3. Що називають лінеаризацією класів?
4. Що називають множинним успадкуванням?

Завдання на самостійну роботу

1. Написати програму, яка відображує успадкування в ООП.
2. Написати програму, яка відображує ієрархію успадкування.
3. Привести приклад програми з лінеаризацією.
4. Привести приклад програми з множинним успадкуванням.

14 ПОЛІМОРФІЗМ

Поліморфізм в інформатиці - це здатність однаковим чином обробляти дані різних типів.

Існує декілька видів поліморфізму. Мови програмування зі статичною типізацією можуть бути статично неполіморфними і статично поліморфними. Для останніх характерні наступні види поліморфізма:

- **спеціальний поліморфізм** - "один інтерфейс - багато реалізацій". Характерні представники: C++, Java, C# - перевантаження (overloading) методів, тобто різні методи з однаковими іменами які приймають параметри різних типів
- **параметричний поліморфізм** - "одна реалізація — багато інтерфейсів". Наприклад, "дженеріки" в Java і C#, шаблони в C++

- **поліморфізм підтипів** - саме те, що розуміють під поліморфізмом в ООП. Ключове поняття: якщо клас В успадковано від класу А, то екземпляр класу В одночасно є і екземпляром класу А.

В мовах з динамічною типізацією поліморфізм присутній завжди, він як би є "побічним ефектом" оскільки при динамічній типізації перевірка типів проводиться вже під час виконання програми.

Поліморфізм ще можна визначити як можливість обробки екземплярів різних типів даних, тобто таких, які створені на основі різних класів, за допомогою функцій (методів) з однаковими іменами. Результати роботи однойменних методів можуть суттєво відрізнятись. В цьому контексті під словом "поліморфізм" можна розуміти "багато форм одного і того ж слова", тобто імені метода.

Ми вже стикались з поліморфізмом між класами, які пов'язані успадкуванням. У кожного класу може бути свій метод __init__() або будь-який інший. Який саме з методів __init__() буде викликано, і що саме він "зробить", залежить від належності об'єкта до того чи іншого класу.

Але класи не обов'язково мають бути зв'язані успадкуванням. Поліморфізм як один з ключових елементів ООП може існувати незалежно від успадкування. Класи можуть бути не пов'язаними ієрархією успадкування, але мати однакові методи, тобто мати однаковий або схожий інтерфейс при зовсім різній реалізації. Створення "єдиних інтерфейсів" дозволяє робити програми більш зрозумілими.

Ключові питання

1. Що називається поліморфізмом в інформатиці?
2. Що таке спеціальний поліморфізм?
3. Що таке параметричний поліморфізм?
4. Що таке поліморфізм підтипів?

ЛІТЕРАТУРА

1. Mark Lutz Learning Python 2019 C-832.

2. Дмитрий Мусин Самоучитель Python М. 2019 С-149.
3. Эл Свейгарт Учим Python, делая крутые игры 2018 С-416.
4. О.Васильєв Програмування мовою Python К. 2019 С-504.
5. https://thecode.media/pygames/
6. https://www.youtube.com/watch?v=wDgZdYRQ4gU.
7. https://pythontutor.ru/lessons/2d_arrays/
8. http://labs.org.ru/python-8/
9. http://progras.ru/31-dvumernye-spiski-massivy-matricy-v-python/
10. https://habr.com/ru/post/246699/
11. https://habr.com/ru/post/478620/
12. https://developer.mozilla.org/uk/docs/Glossary/REST
13. https://uk.wikipedia.org/wiki/REST
14. https://dev.to/code_enzyme/introduction-to-using-async-await-in-python-2i0n https://webdevblog.ru/obzor-async-io-v-python-3-7/