Підручник
з програмування
PYTHON

Basic Python Education

2022 рок.

BPE 

Навчальна програма **Basic Python Education** призначена для вивчення мови програмування Python на базовому рівні при проходження дисципліни “програмування”. Розраховано для студентів 3-го курсу передвищих навчальних закладів, 1-го або 2-го курсу вищіх навчальних закладів за фахом, учнів старшої загальноосвітньої школи та спеціалізованої гімназії або задля самостіного вивчення людьми будь-якого віку.

Навчальна програма складається з наступних матеріалів:

- Електронний підручник з програмування на мові Python
- Набір конспектів лекцій за темами
- Набір задач, запропонованих для самостіного вивчення
- Лабораторні роботи
- Тести для періодичного контролю знань
- Контрольні роботи для загального контролю знань

Повний об’єм матеріалів знаходиться на локальному сайті BPE.

Також сайт містить ряд корисних статей та електронний вигляд тестів та контрольних робот для проходження їх у вигляді тестів Google Form.

ЗМІСТ

Тема 1. Знайомство з Python

Тема 2. Масиви, цикли, функції

Тема 3. Робота з зовнішніми файлами

Тема 4. Об’єктно орієнтоване програмування (ООП)

Тема 5. Фреймовки, Tkinter

Знайомство з Python

Підготовка до роботи з мовою програмування Python

Завантаження Python

Спочатку потрібно завантажити Python з офіційного сайту https://www.python.org/downloads.

Після завантажування запускаємо інсталятор від імені адміністратора. В ході установки встановлюємо прапорець «Додати в PATH».
Чекаємо кінця загрузки та перевіряємо чи вдало пройшла установка.

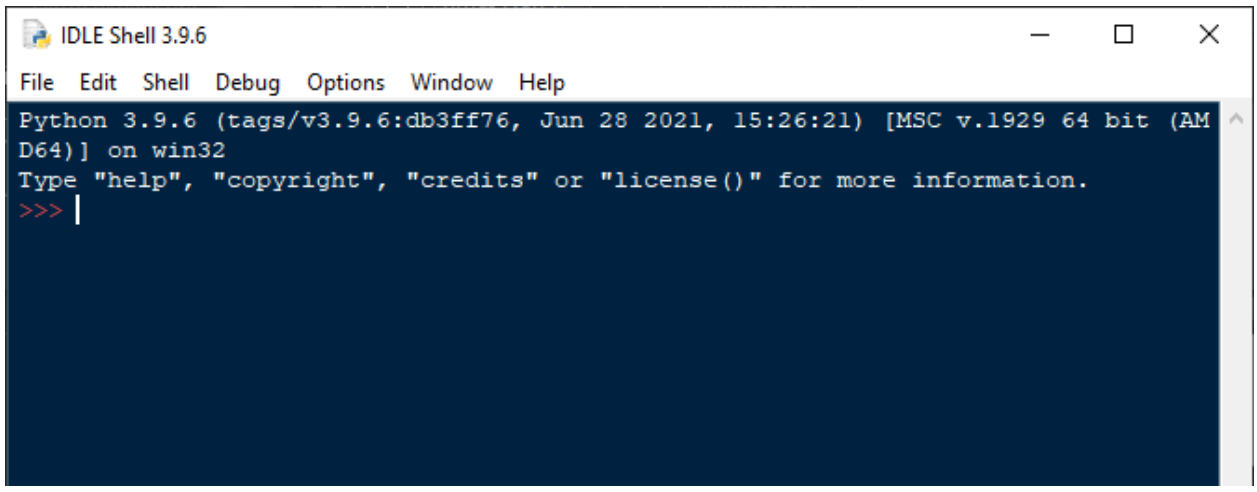
Після встановлення Python, потрібно встановити середовище (IDE – Integrated Development Environment). Для навчання підійде IDLE Python, але буде працювати будь-яке середовище розробки. Так само завантажуємо з офіційного сайту та встановлюємо на комп'ютер.

Робота з IDLE Python

При запуску середовища IDLE автоматично відкривається інтерактивний режим. У цьому режимі команди виконуються одразу після їх введення та підтвердження (*Enter)*.

Головне меню розташовано у вигляді строки над полем ввода та містить такі пункти:

- File – створення, відкриття, збереження файлу та вихід з середовища.
- Edit – редагування файлу.
- Shell – налаштування скриптів.
- Debag – розпознавання помилок програмою пошуку помилок (багів).
- Options – опції середовища.
- Window – налаштування вікна середовища.
- Help – допомога користування середою.



Щоб написати програму в середовищі IDLE потрібно виконати наступні кроки:

1. Відкрити IDLE.
2. Створити новий файл у пункті меню *File* (або *Ctrl+N).*
   Або відкрити вже існуючий файл у тому самому пункті (або *Ctrl+O).*
3. Набрати текст програми.

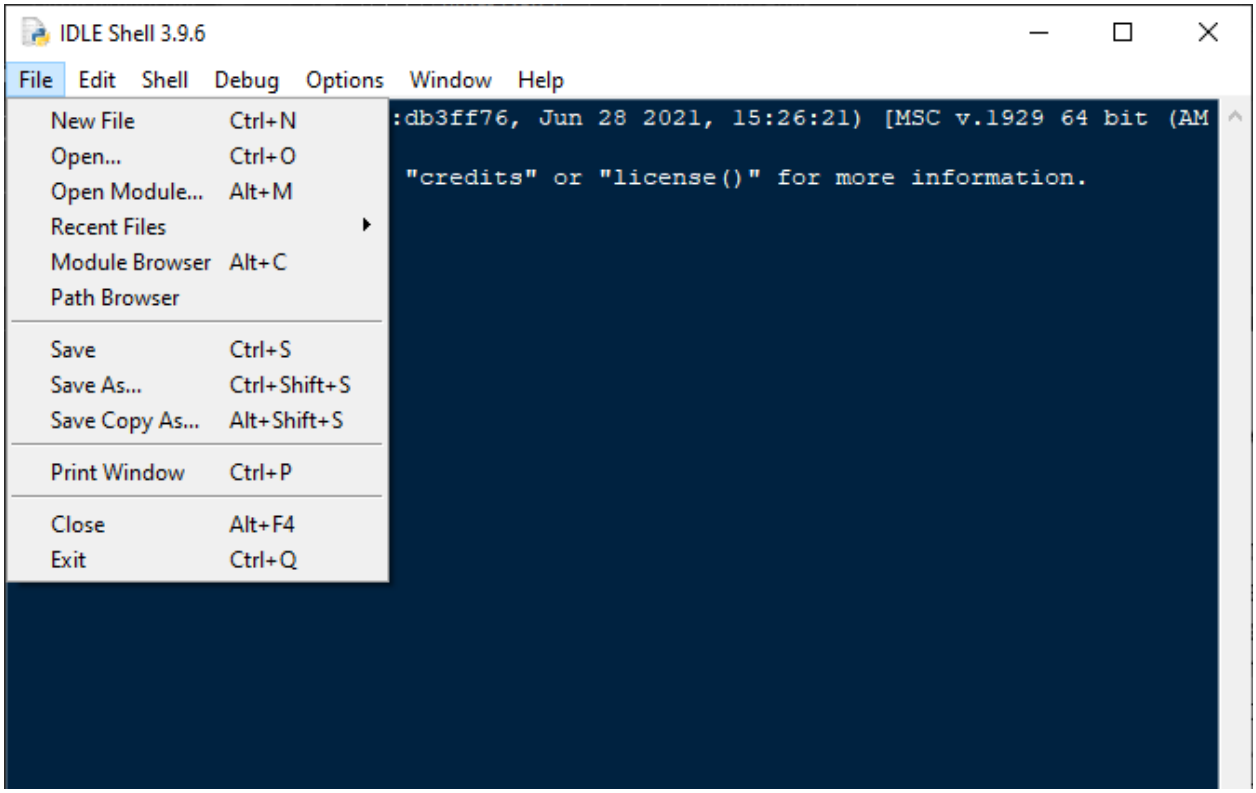
Варто пам'ятати, що Python – мова інтерпретуюча, тобто код при виконанні не компілюється (переводиться до машинного коду цілком), а інтерпретується – переводиться до машинного коду построково, після чого автоматично переходить до наступної строки.

*зауваження: при написанні дробових чисел використовується ТІЛЬКИ крапка.

4. Зберегти програму в меню *File (*або *Ctrl+Shift+S).*
5. Запустити програму.

Якщо вона не має синтаксичних помилок, то відкриється вікно Python Shell, де запуститься написана програма. При відкритті файла окремо від середовища IDLE програма буде запускатися у командній строці операційної системи.

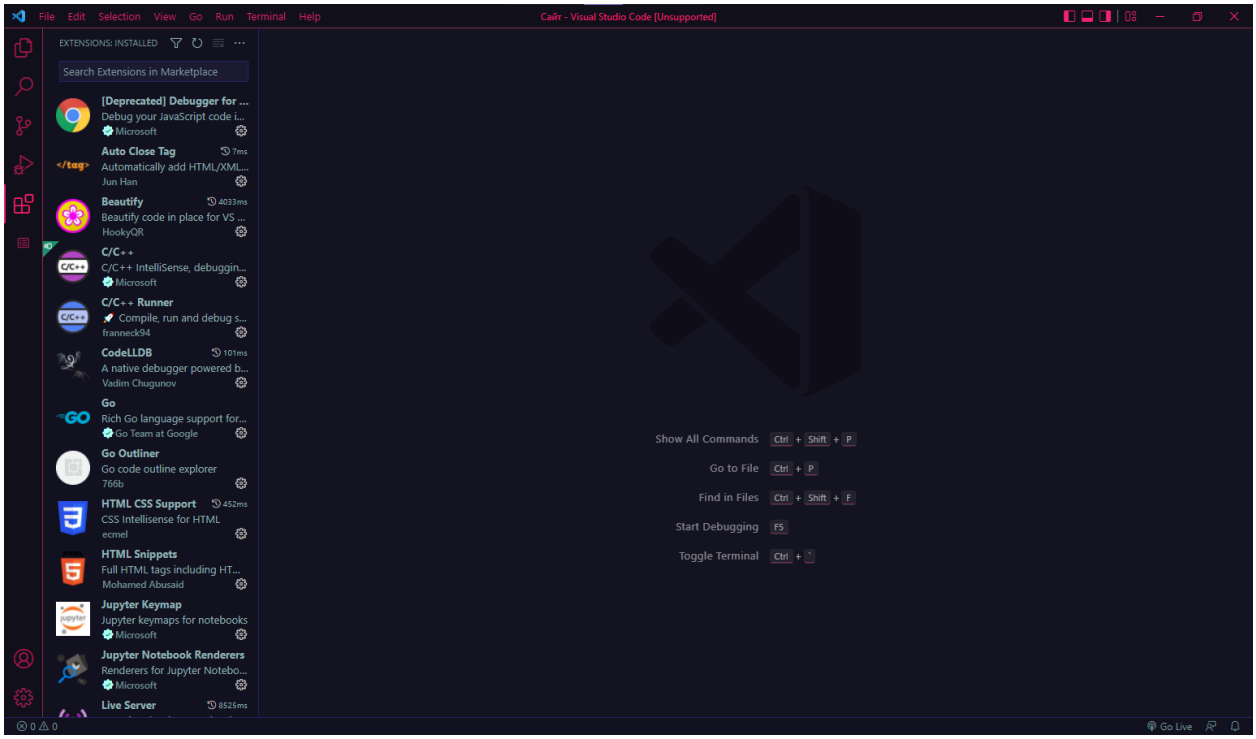
6. Перевірити програму на наявність логічних помилок (тестування).
7. Перед завершенням роботи з середовищем і його закриттям варто перевірити цілісність збереженого файла. Для виходу з середовища можна просто натиснути на хрестик у верхньому правому кутку.



Робота з Visual Studio Code

Visual Studio Code, також відома як VSCode, є дуже легкою для освоєння, порівняно з іншими сучасними IDE, та зручною при користуванні. Також вона майже не навантажує комп'ютер при роботі. Для його роботи також потрібен інтерпретатор Python (посилання – https://www.python.org/downloads).

VSCode можна завантажити безкоштовно за посиланням https://code.visualstudio.com.



Цей редактор надає можливість завантажити та встановити доповнення, які значно полегшують роботу з кодом, відкривати одночасно кілька файлів, контролювати поточну директорію, тестувати та користуватись техлогією Git.

Рекомендовані доповнення для роботи з Python:

- Python
- Pylance
- Code Runner

Корисні горячі клавіши:

- Ctrl+N – створити новий файл
- Ctrl+O – відкрити файл
- Ctrl+Shift+S – зберегти як
- Ctrl+S – зберегти

Робота з мовою програмування Python

Особливість мови Python – це те, що вона використовує інтерпретатор.

Інтерпретатор мови програмування (interpreter) — програма чи технічні засоби, необхідні для виконання інших програм, вид транслятора, який здійснює пооператорну (покомандну, построкову) обробку, перетворення у машинний код та виконання програми або запиту (на відміну від компілятора, який транслює у машинні коди всю програму без її виконання).

Таким чином, Python працює з кодом построково, тому важливо мати наувазі, що порядок написання команд (зокрема, функцій) має значення.

Основний код програми на мові Python, тобто той, що виконується при запуску програми, знаходиться поза функцій (підпрограм). В ньому можуть як і викликатися окремі функції, так і виконуватися команди.

Робота у будь-якій мові програмування відбувається у директорії.

Поточна робоча директорія – невидима змінна яку Python весь час зберігає в пам'яті. Поточна робоча директорія завжди задана, незалежно від вашої діяльності на данний момент.

Для визначення поточної директорії використовується команда os.getcwd().

При запуску Python поточна директорія – це коренева папка, в якій знаходиться Python. Поточну директорію можна змінити на час роботи на папку, у якій знаходиться проект. В цьому випадку інтерпретатор буде враховувати всі файли, що знаходяться у цій папці.

Для невеликих програм, що використовують тільки виконуваний файл, поточну директорію можна не змінювати, але якщо програма включає в себе запуск інших файлів, то варто вказати нову директорію.

Враховуючи всі пункти створення програми, можна написати короткий виконуваний файл. Внегласно традиційним першим скриптом на будь-якій мові програмування є “Hello, world!”. На Python така програма має такий вигляд:

```
print(“Hello, world!”)
```

Створіть таку програму у інтерактивному режимі та окремим файлом.

Прості лінійні алгоритми

Лінійний алгоритм означає, що обчислювальний процес буде виконуватися послідовно, без пропуску яких-небудь частин.

Оператори

Лінійний обчислювальний процес на мові Python передбачає оператори, які виконуються послідовно:

1. Оператор присвоювання =.

Відбувається присвоювання значення змінній. Спочатку виконується операція, що знаходиться справа від оператора присвоювання, а результат виконання записується у змінну, що стоїть зліва.

2. Оператор складання або конкантенації +.

Оператор "+" має два значення, залежно від даних, до яких він використовується. Якщо "+" застосовується до чисел, то відбудеться складання цих чисел; а якщо до строк – конкантенація (об'єднання строк).

Конкатенація — операція склеювання об'єктів лінійної структури. Наприклад, конкатенація слів «мікро» і «скоп» дасть слово «мікроскоп».

3. Оператор множення *.

Оператор множення має також два значення. Він може використовуватися як для двох чисел (множення), так і для числа та строки (повторення строки n кількість разів).

4. Арифметичні оператори -, /, **.

Окрім операторів складання та множення Python має також інші арифметичні оператори, які можуть бути використані тільки стосовно чисел. Це – "-" (віднімання), "/" (ділення) та "**" (возведення у степінь).

5. Логічні оператори **~, >>,** <<, &, |, **^.**

Логічні оператори виконують дію побітово з числом у двійковій системі зчислення.

"~" – інверсія, значення кожного біту числа змінюється на протилежне.

">>" та "<<" – оператори зсуву вправо або вліво на вказану кількість біт.

"&" – бітовий оператор І (AND)

"|" – бітовий оператор АБО (OR)

"^" – бітовий оператор ВИКЛЮЧНЕ АБО (XOR)

Типи данних

У мові Python існує декілька типів данних: цілі числа (*int*), дробові числа (*float*), строки (*str*), логічні вирази (*bool*) та інші, більш складні типи.

Іноді int може бути автоматично переведеним до float (здебільшого – при дії int з float). Також bool може за надібністю бути переведеним у число, наприклад у функції round, (True – 1, False – 0) або число (float або int) до логічного виразу.

Цілі числа можуть взаємодіяти з дробовими та отримувати в результаті дробове число (3+7.2=10.2). Строки можуть бути сполучені тільки з строками (’при’+’віт’=’привіт’). Спробу поєднати число зі строкою інтерпретатор виявить як помилку *ValueError*. Але можливе перетворення одного типу даних в інший. Так, будь-яке число (у тому числі і дробове) можна перетворити у строку. Для цього потрібно виконати наступне:

```
a = 4          #type(a) = int
b = 4.2        #type(b) = float
a = str(a)     #type(a) = str
b = str(b)     #type(b) = str
```

Також можливе перетворення з дробового числа у ціле та навпаки. У першому випадку відбувається «відкидання» дробової частини:

```
b = 4.99     #type(b) = float
b = int(b)   #type(b) = int, b=4
```

У другому випадку число відображається з одним нулем після крапки.

Функції

Окрім операторів дії можуть виконувати команди мови програмування. Для переводу в інші типи даних використовують саме функції.

Функція type() повертає тип даних змінної, що записана в дужках.

Функція print() дозволяє вивести дані на екран.

Функція input() дозволяє користувачеві ввести дані. Повертає строку.

При вводі даних користувачем у функцію *input()*, дані автоматично стають строкового типу. Тому іноді виникає необхідність переводити їх у інші типи. Процедура переводу приведена вище, але для економії об’єму програми можна використовувати наступний вираз:

```
a = int(input('Введіть ціле число'))
```

Таким чином, ми одразу перетворюємо введені дані у ціле число.

Щоб округлити результат обчислень використовують функцію *round()*. В цій команді в дужках пишеться змінна або число, яке потрібно округлити та, через кому, кількість знаків після крапки. Наприклад:

```
a = 7.27494675
a = round(a, 2)  #результат округляється до 2 знаків після коми
```

Умовні конструкції

Множинне розгалуження

Трапляються випадки, коли виконана дія має залежати від початкових даних. Це виконується розгалуженням алгоритму. В таких алгоритмах виконується тільки одна гілка, а інші – пропускаються. На мові Python розгалуження реалізується за допомогою конструкції *if-elif-else*, яка називається *множинним розгалуженням.*

Заголовком називається строка, у якій відбувається перевірка умови (Наприклад: if…:/elif…:/else:).

*Тіло операто*ра – це гілка, по якій виконується програма у разі, якщо перевірка умови поверне значення *“True”*.

Конструкція завжди починається з оператору *if.* При виконанні відбувається перевірка умови, яка записана після оператора.

```
if a >= 0:

if a == 1:

if a < 0:
```

Варто пам’ятати, що “=” – це оператор присвоєння, а “==” – порівняння.

У випадку, якщо оператор if повернув значення *“True”,* то виконується гілка, яка записана в тілі цього оператора. Тіло записується з 4 пробілами (один tab) у кожній строці:

```
if a >= 0:
    a = a+3
    print(a)
```

Для скорочення також можна замінити вираз *a = a+3* на *a += 3*

Якщо оператор if повернув значення *“False”,* то інтерпретатор пропустить тіло цього оператора і перейде до перевірки наступної умови, записаної після оператора elif:

```
if a >= 0:
    a = a+3
    print(a)
elif a < -3:
    a += 1
    print(a)
```

Тіло оператора elif записується також з 4 пробілами. Залежно від кількості умов, операторів elif може бути будь-яка кількість або не бути взагалі.

Залишається останній оператор else. Він виконується тільки в тому випадку, якщо оператор if та всі оператори elif повернули *“False”*. Тіло оператора записується так само з чотирма пробілами:

```
if a >= 0:
    a = a+3
    print(a)
elif a < -3:
    a += 1
    print(a)
else:
    print(a)
```

В наведеному прикладі коді відбувається розгалуження за принципом: якщо а більше або равно 0, то виконується перша гілка, якщо а менша за -3 – друга гілка, а якщо не виконуться обидві умови (0<a<=-3), то просто виводиться а.

Не варто забувати, що оператор if реагує не на дію в його умові, а на результат дії. Наприклад, в умові може бути True – тоді умова буде виконуватися завжди. Також можуть бути наступні трансформації:

True – не порожня строка, будь яке число, окрім 0 та не порожні масиви

False – порожня строка, число 0 та 0.0, None, порожні масиви.

Також є спеціальний оператор, який існує тільки в Python – not. Він позначає інверсію значення, що йде після нього.

Форми запису розгалуджень

Існує кілька способів запису розгалудження:

1) Нормальна (повна) форма:

Зачасту потрібно здійснити вибір по одній умові. Так, якщо не здійснюється умова, програма програє гілку else.

Синтаксис:

```
>>> if a < 50:
        print(a)
else:
        print(a-50)
```

Для написання кількох умов використовуматися наступна конструкція:

```
if a < 0:
    print(a)
else:
    if a == 0:
            print(a)
    else:
            if a > 0:
                    print(a)
            else:
                    pass
```

Такий спосіб перевірки багатьох умов одночасно вважається дуже об’ємним та довшим у виконанні в порівнянні з if-elif-else.

2) Коротка форма запису (*тернарний вираз*):

Це э короткою формою запису для тіх самих цілей:

Синтаксис:

```
>>> print(a) if a < 50 else print(a-50)
```

Зазвичай так перевіряють одну умову, але також є можливість перевірки кількох умов.

```
>>> print(a) if a<50 else print(a-50) if a<70 else print(a-70)
```

Стане зрозуміліше, якщо додати у вираз круглі :

```
print(a) if a<50 else (print(a-50) if a<70 else print(a-70))
```

3) If-elif-else (еквівалент switch case):

Для перевірки багатьох умов використовують switch case. У Python його немає, але є аналог, який діє так само.

Синтаксис:

```
if a >= 0:
    a = a+3
    print(a)
elif a < -3:
    a += 1
    print(a)
else:
    print(a)
```

Бібліотеки, зовнішні модулі

Імпорт бібліотек (модулів), модуль MATH

Для спрощення життя програміста існують модулі (бібліотеки). Найпоширеніші з них вбудована в інтерпретаторний пакет Python. Для визову таких достатньо їх імпортувати. Якщо Ви бажаєте використати зовнішній модуль, Вам потрібно скачати його та занести до списку бібліотек або скористуватися файл-менеджером PyInstaller (pip).

Імпортувати модуль означає додати до основних функцій набір додаткових, які містить цей модуль. Це можна зробити чотирма способами:

1) Імпорт всієї бібліотеки:

```
import math
```

При цьому визов функції потребує указання назви модуля:

```
b = math.sin(a)
```

2) Імпорт всієї бібліотеки під іншою назвою:

```
import math as ma
b = ma.sin(a)
```

Визов функції потребує указання нової назви.

3) Імпорт окремої функції з бібліотеки:

```
from math import sin()
b = sin(a)
```

Визов функції не потребує указання назви модулю, але, якщо існувала функція з такою самою назвою, то вона перезаписується на нову.

Можна імпортувати окрему функцію під іншою назвою. В такому випадку виклик потребує вказання нової назви.

4) Імпорт всіх функцій бібліотеки:

```
from math import *
b = sin(a)
```

Імпортуються всі функції, при визові не треба вказувати назву модуля, але недолік той самий, як і при імпортуванні окремих функцій.

Але у цього метода є додаткова перевага: у програмі не будуть записані додаткові функції, які не використовуються. Це може вплинути на продуктивність програми та зменшити її обсяг.

Модуль math

У модулі math є такі функції:

· тригонометричні функції: Sin, Cos, Tan;

· зворотні тригонометричні функції: Asin, Acos, Atan, Atan2;

· гіперболічні функції: Tanh, Sinh, Cosh;

· експоненту і логарифмічні функції: Exp, Log, Log10;

· модуль (абсолютну величину), квадратний корінь, знак: Abs, Sqrt, Sign;

· округлення: Ceiling, Floor, Round;

· мінімум, максимум: Min, Max;

· ступінь, залишок: Pow, Ieeereminder;

· повне множення двох цілих величин: Bigmul;

· ділення і залишок від ділення: Divrem.

Крім того, у класу є два корисні поля: число і число e.

Для нас основними на даний момент будуть

ceil()– округлення в більшу сторону,

floor() – округлення в меншу сторону,

sin() – сінус радіану,

cos() – косінус радіану,

tan() – тангенс радіану,

radians() – перевод з градусів в радіани.

Деякі функції дублюють вже вбудовані. Наприклад, функція pow() бібліотеки може замінитися оператором **. Також abs(), min() та max() просто перезаписуються. Значних змін функціонал не зазнає, але у випадку з min та max вони стають більш вузьконаправленими. Так, до перезапису функцій вони могли застосовуватися до масивів та строк. Після перезапису вони стали інтерфейсом між числами. Min() та max() можуть приймати необмежену кількість аргументів.

Масиви, цикли, функції

Списки та кортежі

Масив – це структура даних, що містить у собі прості типи даних, такі як строки, цілі або дробові числа. У Python немає обмежень по входженню типів данних у масив, в той час як в більшості інших мовах програмування дані в одному списку мають бути виключно одного типу. Це означає, що у одному масиві можуть знаходитися як строки, так і числа, так і логічні вирази.

Мова програмування Python має 4 основних вида масивів: списки, кортежі, словники та множини.

Списки

Найпростіший спосіб *створити список* – це перелічити всі елементи через коми всередені квадратних дужок або створити пустий список указанням квадратних дужок.

```
spysok = ['a', 4, -3.8, b, '8k']
```

```
new_list = []
```

Варто зазначити, що список не може називатися list, тому що це є назвою команди конвертації деякого типу даних до списку.

Кожен елемент списку має свій індекс. Індексація елементів починається з 0. Також у Python існує зворотня індексація. Елемент з індексом -1 буде останнім в масиві, з індексом -2 – передостаннім, і так далі.

```
>>> arr = [1,2,3,4,5,6,7,8,9]
>>> arr[4]
5
>>> arr[-4]
6
```

Для кожного масиву можна зробити *зріз* – виділена частина масиву або весь масив, котрий був обраний для оперування. Для цього треба надати новій (або вже існуючій) змінній форму типу [*:*], де * - це індекс крайніх елементів. Перший вказанний індекс входить в зріз, а останній – ні.

Якщо потрібно занести в зріз останній елемент масиву, то пишеться [*:] (* - індекс першого елементу).

Якщо зріз повинен дорівнювати всьому масиву, то запис матиме вигляд [:].

```
>>> b = 9.2
>>> spysok = ['a', 4, -3.8, b, '8k']
>>> a = spysok[0:3]
>>> a
['a', 4, -3.8]

>>> b = spysok[3:]
>>> b
[9.2, '8k']
```

Списки є змінними структурами. Це виявляється у тому, що можна змінити елемент списку, видалити його або додати новий.

Зміна елемента відбувається за його індексом:

```
>>> spysok[3] = 7
>>> spysok[3]
7
```

Видалення відбувається відповідно за індексом або за їх значенням:

```
>>> spysok.pop(3)
7
>>> spysok
['a', 4, -3.8, '8k']

>>> spysok.remove('a')
>>> spysok
[4, -3.8, '8k']
```

Щоб додати в кінець списку елемент потрібно використати функцію:

```
>>> spysok.append('cor')
>>> spysok
['a', 4, -3.8, '8k', 'cor']
```

Щоб додати елемент всередину списку використовується функція insert, де в параметрах записується на першій позиції індекс, який буде присвоєний елементу та сам елемент через кому:

```
>>> spysok.insert(2, 'bren')
>>> spysok
['a', 4, 'bren', -3.8, '8k', 'cor']
```

Кортежі

Кортеж – це такий самий список, але, на відміну від нього, це незмінна структура даних. Він записується в круглих дужках.

```
kortej = ('a', 4, -3.8, b, '8k')

kortej = ()
```

Функцією конвертації в кортеж є tuple(). Відповідно, масив не може називатися як tuple.

Кортеж неможливо змінити, але все одно можна зробити зріз структури.

Також у всіх типах масивів можна складати зрізи:

```
>>> kortej2 = kortej[:2]+kortej[3:]
>>> kortej2
('a', 4, 9.2, '8k')
```

Для взяття елементу кортежа по індексу використовують таку ж саму дію, як і для списку:

```
>>> kortej[2]
-3.8
```

Словники та кортежі

Словники

Словник – це невпорядкована змінна структура даних. У елементів словника немає індексів, але вони мають ключи. Словники записуються у фігурних дужках за типом: “ключ : елемент”.

```
slov = {'2':'dog', '1':'cat', '4':'hamster', '7':'crocodile'}
```

Функція конвертації – dict().

Пошук по словнику відбувається наступним чином:

```
>>> slov['4']
'hamster'
```

Так як словники є змінними, то в них можна добавити, видалити або змінити елемент.

Щоб додати елемент:

```
>>> slov['5']='elephant'
>>> slov['5']
'elephant'
```

Змінити його:

```
>>> slov['7']='zubastyk'
>>> slov['7']
'zubastyk'
```

Або видалити:

```
>>> del slov['5']
>>> slov
{'2': 'dog', '1': 'cat', '4': 'hamster', '7': 'zubastyk'}
```

У одному словнику не може бути двох елементів з однаковими ключами, але можуть бути однакові елементи, які належать різним ключам.

У випадку, якщо у одному словнику буде 2 однакових ключа, то ключ, що був записаний першим, видаляється разом з елементом.

Множини

Множини – це масиви неповторюваних невпорядкованих елементів. При конвертації, наприклад, списку, усі дублікати елементів видаляються автоматично. Множини можуть бути змінюваними (set) або незмінюваними (frozenset). Їхня різниця аналогічна до різниці між списком та кортежем.

Множини записуються у фігурних дужках, але створення пустої множини записом дужок неможливе, т.я. в такому випадку створиться

порожній словник. Множина записується за допомогою конвертації списку, кортежу або строку або вказанням порожньої функції:

```
>>> array = set()
>>> array
set()
```

```
>>> array = set([1,3,5,4,4,7])
>>> array
{1, 3, 4, 5, 7}
```

При вводі рядка у функцію set() створиться множина унікальних символів цього рядка:

```
>>> array = set('hello, world!')
>>> array
{'w', 'r', 'd', 'l', 'h', 'e', ',', ' ', '!', 'o'}
```

Для додання до множини елементу використовується метод set.add(), в дужках якого вказують елемент, що додається:

```
>>> array = set()
>>> array
set()
>>> array.add(4)
>>> array
{4}
```

Для видалення елементу використовують один з наступних методів:

- set.remove() – прибирає елемент, зазначений в дужках. Помилка (KeyError), якщо елемента немає в множині.
- set.discard() – прибирає елемент, зазначений в дужках. Якщо елемента немає, не повертає помилку.
- set.pop() – прибирає випадковий елемент з множини;
- set.clear() – очищує множину.

Також над множинами можливо виконувати логічні операції:

- & – логічне І;
- | – логічне АБО;
- - – логічне віднімання;
- ^ – імплікація.

Для змінення вже існуючої множини використовують ті самі дії, але зі знаком =:

```
>>> array = {4,5,6}
>>> arr = {5,6,7,8}
>>> array &= arr
>>> array
{5, 6}
```

```
>>> array |= arr
>>> array
{5, 6, 7, 8}
```

Хоч freezeset змінити неможливо, але з ними все також можна виконувати логічні операції. При цьому початкові множини залишаються незмінні.

Генератори масивів

Генератори списків

В мові програмування Python для створення та заповнення елементами масивів є значне спрощення. Для цих цілей використовують генератори списків (*list comprehension).*

Генератор списків дозволяє швидко створити та заповнити список на базі іншого масива. Генератор списку записується у квадратних дужках та має наступні частини:

```
>>> a = [1,2,3]
>>> b = [i+10 for i in a if i<=2]
```

1) Дія, що виконується з кожним елементом перед записом у список. (i+10)
2) Назва змінної. (for i)
3) Початковий (базовий) масив. (in a)
4) Можлива наявність умови. (if i<=2)

Умова повинна бути у вигляді тернарного виразу, але її присутніть не обов'язкова взагалі (у такому випадку будуть занесені всі перелічені елементи).

Також в якості початкового 22ассива може використовуватися *діапазон*.

```
>>> b = [i+10 for i in range(6)]
>>> b
[10, 11, 12, 13, 14, 15]
```

Так як списки майже однакові з кортежами, то після використання генератора списка результат можна конвертувати у кортеж.

```
>>> b = tuple([i+10 for i in range(6)])
>>> b
(10, 11, 12, 13, 14, 15)
```

Генератори словників та множин

Для створення заповнених словників та множин також є генератори. Принцип їх написання не відрізняється від генераторів списків, але квадратні дужки замінюються на фігурні.

```
>>> a = [1,2,3,4,5]
>>> b = {i:i*2 for i in a}
>>> b
{1: 2, 2: 4, 3: 6, 4: 8, 5: 10}
```

Для створення словників потрібно вказувати пару ключ-елемент. Вони вказуються через двокрапку.

```
>>> a = [1,1,2,2,3,4,5,5,5]
>>> b = {i for i in a}
>>> b
{1, 2, 3, 4, 5}
```

Функції map() та lambda

Для створення простих можна використовувати *анонімну функцію* lambda.

Приклад функції lambda:

```
lambda x: x*2
```

В данному випадку функція приймає x та вертає його подвійне значення.

Для швидкої обробки вже існуючих списків існує функція map(). В її параметри вводяться функція, яка застосовується на кожен елемент масиву послідовно, і, власне, сам список, що оброблюється.

```
>>> a = [1,2,3,4,5]
>>> b = list(map(lambda x: x*2, a))
>>> b
[2, 4, 6, 8, 10]
```

Функція map() повертає об’єкт, тому для його корректного читання потрібно конвертувати результат у список.

Цикл while та for

За допомогою циклів можна створити багаторазове повторення частини кода. Така необхідність виникає дуже часто. Наприклад, якщо потрібно знайти суму чисел в масиві.

Усього в Python є два типа циклів: while та for.

Цикл While

“While” перекладається як «поки», а конкретніше, то «поки виконується умова, виконуємо це». Власне, це і є сутність цього типа циклу.

Цикл while має заголовок та тіло. У заголовку перевіряється умова. Якщо вона повертає значеняя True, виконується код тіла. Після завершення останньої команди тіла циклу, інтерпретатор повертається до заголовку та робить перевірку заново. Таке коло буде відбуватися до тих пір, поки з заголовка не повернеться значення False. Дуже важливо *надавати можливість змінній, по якій йде перевірка, змінювати своє значення*. Якщо заголовок буде весь час повертати істину, то цикл стане нескінченним. Іноді це робиться спеціально, але при деяких умовах цикл все одно завершується.

Приклад циклу:

```
a = int(input('Число '))
while a < 100:
    a += 10
    print(a)
```

Та його виконання:

```
Число 33
43
53
63
73
83
93
103
```

Кожне виконання тіла циклу називається *ітерацією.* Цикл вище виконав 7 ітерацій, перше ніж повернути False.

Цикл for

Цикл for у Python призначений для перебора елементів структур даних (наприклад, списків). *Перебор елементів структури даних*– це послідовне вийняття кожного елементу структури та дії з ними.

У заголовку цикла for знаходиться масив на назва *локальної* змінної, яка приймає значення кожного з елементів массива послідовно з кожною новою ітерацією. Після завершення циклу ця змінна зчезає.

```
spis = [7, 3.42, -14, 0]
for i in spis:
```

Тіло цикла for – це набір команд, які виконуються з кожним елементом массива послідовно.

Приклад циклу for:

```
spis = [7, 3.42, -14, 0]
for i in spis:
    i += 2
    print(i)
```

Результат виконання циклу:

```
9
5.42
-12
2
```

При цьому у самому списку зміни не зберігаються:

```
>>> print(spis)
[7, 3.42, -14, 0]
```

Буквою «i» ми позначаємо елемент списка. Кожна наступна ітерація буде брати елемент з індексом на 1 більший за попередній. Замість «i» може бути будь-яка літера або сполучення літер (нова змінна).

Можемо замінити цей цикл більш складною конструкцією, яка містить while:

Цикл:

```
spis = [7, 3.42, -14, 0]
i = 0
while i < len(spis):
    print(spis[i]+2)
    i += 1
```

Результат виконання циклу:

```
9
5.42
-12
2
```

Функція *len()* визначає кількість елементів у структурі даних.

Цей метод також не змінює початковий список і є рівносильним данному циклу for. Внесення змін у сам масив розглядатиметься пізніше. Він можливий як для циклу for, так і для while.

Break, continue

Для виходу з циклу використовується інструкція *break*. Зачасту вона знаходиться в тілі оператора if або else.

```
i = 1
while(True):
    if i > 10:
        break
    else:
        i += 1
```

Для пропуску ітерації використовується інструкція *continue*. Зазвичай вона також знаходиться в тілі оператора if або else.

```
i = 1
arr = []
while i < 10:
    i += 1
    if i % 2 == 1:
        continue
    arr.append(i)
```

Діапазони, функції range, enumerate

Функції range(), enumerate()

Функція *range()* працює наступним чином: вона приймає від одного до трьох чисел та робить діапазон послідовних чисел. Якщо введене одне число, то границею автоматично стане 0. Якщо два числа – діапазон буде будуватися від першого до другого, при цьому перше число обов'язково повинне бути менше за друге. Третій параметр – це шаг (за замовченням = 1).

Структура даних, яку створює функція, називається *діапазоном*. Ця структура не є змінною та виводиться в термінал як *range(*,*)*.

При написанні границь варто враховувати, що нижня границя входить до діапазону, а верхня – ні. Таким чином, range(5, 8) буде відповідати 5,6,7.

Діапазони є ітеруємою та впорядкованою структурою данних, тобто їх можна перебрати по порядку за допомогою цикла for.

Функцію range() дуже ефективно поєднувати з функцією len(). За допомогою отриманного діапазону від 0 до кінцевого індекса 26ассива можна змінити кожет елемент списку за допомогою цикла for:

```
spis = [7, 3.42, -14, 0]
for i in range(len(spis)):
    spis[i] += 2
print(spis)
```

Результат виконання циклу: `[9, 5.42, -12, 2]`

Таким чином, цикл for перебирає не елементи списка, а їх індекси.

Якщо range() дає нам можливість оперувати індексами списка, то функція enumerate() генерує кортежи з парами індекс-елемент.

Цикл:

```
spis = [7, 3.42, -14, 0]
for i in enumerate(spis):
    print(i)
```

Результат виконання циклу:

```
(0, 7)
(1, 3.42)
(2, -14)
(3, 0)
```

Ці кортежі можливо розпаковувати. Для цього, зазвичай використовують дві змінні у заголовку цикла for:

Цикл:

```
spis = [7, 3.42, -14, 0]
for i, k in enumerate(spis):
    print(i, k)
```

Результат виконання циклу:

```
0 7
1 3.42
2 -14
3 0
```

Виключення в мові Python

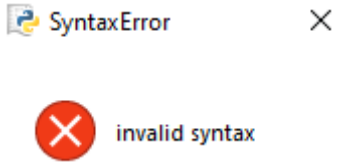
У будь-яких мовах бувають помилки. Помилки діляться на *синтаксичні* (в результаті програма не працює взагалі) та *логічні* (в результаті програма працює, але робить не те, що було заплановано).

Виникнути помилка може з ряду причин.

Синтаксичні помилки (*виключення*) бувають наступні:

SyntaxError

Власне, це і є випадок, коли виявляється неправильне написання змінної (наприклад, вона починається з числа або з якогось спецсимволу).



ValueError

Виникає при спробі перевести змінну у неприпустимий для неї формат (наприклад, перевести строку з буквами у значенні у число).

```
Traceback (most recent call last):
  File "C:/Users/Magnum/Desktop/ddd.py", line 2, in <module>
    stre = int(stre)
ValueError: invalid literal for int() with base 10: '2k'
```

ZeroDivisionError

Виникає при спробі поділити на нуль.

```
Traceback (most recent call last):
  File "C:/Users/Magnum/Desktop/ddd.py", line 2, in <module>
    stre = int(stre)/0
ZeroDivisionError: division by zero
```

TypeError

Виникає при виконанні дії з операндами неприємлемого типу (наприклад, підвести число у степінь змінної, значення якої – строка).

```
Traceback (most recent call last):
  File "C:/Users/Magnum/Desktop/ddd.py", line 2, in <module>
    stre = stre/2
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

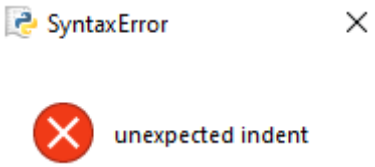
NameError

Виникає при спробі використання змінної, значення якої не вказувалось раніше.

```
Traceback (most recent call last):
  File "C:/Users/Magnum/Desktop/ddd.py", line 3, in <module>
    print(st)
NameError: name 'st' is not defined
```

IndentationError

Виникає при вказанні неправильного отступу.



IndexError

Виникає при вказанні недійсного для списку або кортежу індексу.

```
Traceback (most recent call last):
  File "C:/Users/Magnum/Desktop/ddd.py", line 2, in <module>
    print(stre[5])
IndexError: list index out of range
```

KeyError

Виникає при вказанні неіснуючого ключа для словника.

```
Traceback (most recent call last):
  File "C:/Users/Magnum/Desktop/ddd.py", line 2, in <module>
    print(stre[5])
KeyError: 5
```

Якщо знати що означають виключення та вміти їх читати, то можна доволі швидко привести код у робочий стан.

Також з помилкам у Python можна працювати. Більш того, є інструкція raise, яка піднімає штучно виключення. У інструкцію можна передати назву помилки, щоб отримати користувацьке виключення.

```
>>> raise CustomError
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    raise CustomError
NameError: name 'CustomError' is not defined
```

Таким чином можна шукати помилки помодульно.

Кострукція try-except

Окрім raise для роботи з виключеннями є також конструкція try-except.

При виникнені виключення програма закривається аварійно. Так як Python виконується построково, помилка може не бути виявлена одразу. На випадок, якщо є шанс виникнення якоїсь з помилок, існує конструкція *try-except.*

Вона складається з двох гілок:

Try – спробувати виконати ділянку коду. Якщо виникає помилка, то інтерпретатор автомматично переходить до гілки *except*. А якщо гілка try виконалась без помилок, то гілка except пропускається.

Приклад поєднання try-except з циклом while:

```
a = input('Введіть чило ')
while type(a) == str:
    try:
        a = float(a)
    except:
        a = input('Введіть число, а не букву ')
```

Спочатку ми даємо можливість ввести число та отримуємо строку у будь-якому випадку. Після чого відбувається вход у цикл, так як в перший раз умова буде вірною у будь-якому випадку. В гілці try ми спробуємо перевести строку до числа, а якщо буде ValueError – примушуємо користувача вводити строку до тих пір, поки виключення не зникне. Таким чином ми виключили варіант аварійного закриття через користувача.

Можна зробити так, щоб гілка except реагувала на якийсь конкретний тип виключень. Тоді при різних помилках будуть активуватися різні прописані дії. Для позначення окремого типу виключення пишемо:

```
except ValueError:
```

Замість ValueError може стояти будь-який тип (у тому числі і користувацьки виключення, підняті за допомогою raise).

Якщо до конструкції try-except додати else, то гілка else виконається у випадку, якщо гілка try завершиться без помилок. Також можна додати гілку finally, яка буде запускатися у будь-якому випадку, навіть якщо програма закриється аварійно.

```
try:
    a = 10
    if a>0:
            raise ValueError
except:
    print('a>0')
    raise AMoreThanZeroError
else:
    print('a<=0')
finally:
    print('end')
```

Вивод:

```
a>0
end
Traceback (most recent call last):
  File "<pyshell#16>", line 4, in <module>
    raise ValueError
ValueError
```

Власні функції

Створення функції

В Python є можливість робити *функції* (*підпрограми*). Вони використовуються, якщо потрібно виконати одні й ті самі дії з різними даними в різних місцях програми.

На блок-схемі така підпрограма виписується окремою схемою, а її виклик – прямокутником з чотирма вертикальними сторонами.

Так як Python – це інтерпретована мова, то виклик функції повинен бути після її об`явлення. Зазвичай це робиться у самому початку кода. Іноді одна функція може мати виклик іншої. В данному випадку їх порядок не принциповий.

Об`явлення функції має вигляд: ***def*** *назва функції****():***

Наприклад:

```
def test():
    a = input('')
```

Далі, на відстані 4 пробілів (одного tab) пишеться текст функції. В данному прикладі функція має назву test. Функція завжди визивається однойменно. Вона тільки запрашує один ввод і нічого більше не робить.

Написання функції називається її *декларуванням.*

Локальні та глобальні змінні

Глобальною називається *змінна*, яка вводиться у основному коді та редагується або використовується у ньому. Вони не стосуються кодів підпрограм (функцій).

Функція має «свої» змінні – *локальні*. Вони можуть мати таку саму назву, що й змінні в основному коді, але перекликатися вони не будуть. По закінченню виконання функції всі локальні змінні зникають. Для того, щоб зберегти якусь змінну з функції та імпортувати її в основний код, використовують команду *return()*. Ця команда зупиняє виконання коду функції та доставляє змінну, зазначену в дужках команди, у основний код.

Наприклад:

```
def inp():
    a = int(input('Введіть a '))
    b = (a*3)+8
    return(b)
```

Данна функція запитує у користувача число a, переводить це значення у число та виконує над ним дії. Після цього функція повертає кінечне значення у основний код програми. Щоб прийняти значення локальної змінної у основному коді, потрібно присвоїти це значення глобальній змінній. Тобто при виклику функції ми прописуємо такий напис:

```
a = inp()
```

У випадку, коли потрібно повернути більше однієї змінної, у команді return() вказуються ці змінні через кому у круглих дужках.

Якщо потрібно виконати дію зі змінними з основного коду за допомогою функції, то у скобках при назві функції вказуються локальні змінні, які будуть оброблятися, а при визові у тому ж порядку присваювати їм значення глобальних змінних. Передані змінні мають назву *параметри.*

Наприклад:

При написанні функції

```
def test(a,b,c):
```

При виклику функції (повернену змінну присвоїти глобальній змінній a):

```
a = test(per,dwa,tri)
```

Значення параметра за замовченням

Значення параметра може бути вказане за замовченням. Для цього при написанні функції треба присвоїти параметру значення. Якщо при виклику функції буде вказане інше значення, то буде використане останнє. Наприклад:

```
def test(a = 3):
    return a**2

print(test())       # Буде 9
print(test(2))      # Буде 4
```

Зовнішні користувацькі бібліотеки

Зовнішні модулі

Окрім вбудованих бібліотек в Python можна завантажити *зовнішні модулі.* Вони існують для більш швидкого та легкого досягнення окремих цілей. Наприклад, модуль cv2 існує для роботи з веб-камерою, flask – для веб-розробки та т.і.

Для завантаження зовнішніх модулів можна скористатись файловим менеджером. Файловий менеджер – комп'ютерна програма, що надає інтерфейс користувача для роботи з файловою системою та файлами. Найпоширенішім при роботы з Python є *pip*. Після його завантаження на ПК, для встановлення будь-якої бібліотеки достатньо лише в консолі операційної системи написати команду:

pip install *назва модуля*

та дочекатись завершення завантаження.

Після загрузки бібліотеки на комп'ютер її можна імпортувати у python файл як звичайний модуль.

Корисні вбудовані функції

Розглянемо деякі функції для роботи з символами та деякі математичні функції, що знаходяться не в модулі math.

Функція *ord()* (Order) приймає в якості аргумента один символ в лапках та повертає його номер у таблиці Unicode.

```
>>> ord('r')
114
>>> ord('9')
57
>>> ord('%')
37
>>> ord('y')
1091
```

Функція *chr()* (Character) приймає в якості аргумента ціле число та передає символ з таблиці Unicode за цим номером.

```
>>> chr(10)
'\n'
>>> chr(114)
'r'
>>> chr(37)
'%'
```

Функція len(), окрім масивів, може також приймати строки. Результатом буде кількість символів у данному рядку.

```
>>> len('qw_er ty')
8
>>> a = 'string'
>>> len(a)
6
```

Щодо математичних функцій, не завди доцільно викликати модуль math, так як існують вбудовані аналоги функцій.

Функція *abs()* приймає число на повертає його абсолютне значення, тобто його значення по модулю.

```
>>> abs(-17.51)
17.51
>>> abs(17.52)
17.52
```

Функція *round()* округляє отримане число до вказаного знаку. Першим аргументом функція приймає число, другим – знак після дробу, до якого буде округлення. Якщо аргумент буде всього один, то число буде округлятися до цілої частини. Також другий аргумент може бути від'ємним – тоді число буде округлятися до десятків/сотень/тисяч/т.д.

```
>>> round(12.358936, 2)
12.36
>>> round(-17.94)
-18
>>> round(15743, -2)
15700
```

Функція *divmod()* приймає два числа та повертає кортеж з двох елементів – результта ділення націло першого аргумента на другий та залишок від їх ділення.

```
>>> divmod(27, 5)
(5, 2)
```

Аналогом цієї функції є два оператора: // (ділення націло) та % (залишок від ділення націло).

```
>>> 27//5
5
>>> 27%5
2
```

Функції max() та min() можуть працювати з нескінченною кількістю аргументів. Перша повертає серед них найбільший, друга – найменший. При цьому обидві ці функції можуть приймати строки як аргумент. У такому випадку сортування відбуватиметься не за значенням, а за алфавітом. Також в функції можливо передавати переліки елементів. У цьому випадку результатом є відповідний елемент масиву.

```
>>> max(4,2,12)
12
>>> min(4,2,12)
2

>>> max('abc', 'bcdef', 'f')
'f'
>>> min('100', '1', '01')
'01'

>>> s = [1,2,3,4,5]
>>> max(s)
5
>>> min(s)
1
```

Функція *sum()* може отримувати тільки перечень елементів і тільки з числами. Повертає функція суму всіх елементів переданного масиву.

```
>>> s = [1,2,3,4,5]
>>> p = (0, 12, 23, 20)
>>> sum(s)
15
>>> sum(p)
55
```

Створення власної бібліотеки

Для створення власного модулю робимо файл з розширенням *.py*, який буде складатись тільки з функцій. Назва цього файлу – це назва , за якою її можна імпортувати. А назви функцій, записаних у цьому файлі – це назви функцій модулю. Після того, як наша бібліотека буде цілком готова, зберігаємо її в ту саму папку, де знаходиться інтерпретатор мови Python (там же знаходяться всі інші бібліотеки) або у папку проекта (в цьому випадку цим модулем можна користуватися лише в рамках директорії проекту).

Власні бібліотеки можна імпортувати до файлу так само, як і вбудовані додаткові бібліотеки.

Якщо у власній бібліотеці, окрім функцій, буде міститися власний виконуваний код, то він буде також імпортуватись у до інших файлів та програватись у місці імпорту.

Багатомірні масиви, рекурсія

Створення багатомірного масиву та його індексація

Масиви в Python являють собою набір посилань на елементи, які збережені окремо. Тому в масиві можуть бути будь-які типи даних одночасно. За тим самим принципом у масиві можна залишити посилання на інший масив, при тому не важливо, буде це список, кортеж або словник.

При вкладенні масивів один в одний масив, у котрий вкладають, буде називатися *головним* або *батьківським*. А масив всередені батьківського – *вкладеним* або *дочірнім*. Такі назви працюють на будь-якому рівні вкладеності, відносно своїх батьківских масивів.

```
>>> arr1 = [1,2,3]
>>> arr2 = (2,4,7,'k')
>>> main_arr = [arr1, 5,6,arr2,'q']
>>> print(main_arr)
[[1, 2, 3], 5, 6, (2, 4, 7, 'k'), 'q']
```

Таким чином, ми можемо взяти масив за його індексом у головному масиві.

```
>>> print(main_arr[3])
(2, 4, 7, 'k')
```

Якщо нам потрібно взяти конкретний елемент зі вкладеного масиву, то після вказання індексу дочірнього масиву вказуємо індекс елементу, що нам потрібен, у нових квадратних дужках.

```
>>> print(main_arr[3][2])
7
>>> print(arr2[2])
7
```

Приведені в прикладі вирази є рівносильними, так як кортеж arr2 вкладений в main_arr на 4-ту позицію (має індекс 3).

Якщо всередені вкладеного масиву є ще один масив, ми можемо достати з нього елемент, вказавши ще один індекс в окремих квадратних дужках. Така дія розповсюджується на нескінченну кількість вкладених масивів.

Будь-які масиви, в тому числі і багатомірні, підтримують зворотню індексацію:

```
>>> print(main_arr[-2][-2])
7
```

Дії з багатомірними масивами

Для перебору масивів доцільніше використовувати цикл for.

Якщо просто перебрати масив як одномірний, то програма поверне вкладені масиви.

```
>>> arr = [[1,2,3,4],[5,6],(7,8,9)]
>>> for i in arr:
        print(i)


[1, 2, 3, 4]
[5, 6]
(7, 8, 9)
```

Але можливо всередені одного циклу зробити ще один, щоб він робив перебор вкладеного масиву. Такий цикл аналогічно називається *вкладеним*.

```
>>> for i in arr:
        if type(i) == list or type(i) == tuple:
                for k in i:
                        print(k)
        else:
                print(i)


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

Данний приклад виводить кожен елемент дочірніх масивів. Також у ньому робиться перевірка на випадок, якщо у батьківському списку окрім масивів є неітеруємий елемент. У випадку, якщо цикл for отримає такий елемент, відбудеться помилка TypeError.

Рекурсія

Також перебор та вивод елементів можна виконати рекурсивною функцією. Рекурсивна функція лежить в основі стеку.

Стек – структура данних, яка працює за принципом «останнім прийшов — першим пішов».

Рекурсивна функція – це функція, яка викликає саму себе в процессі виконання. При створенні рекурсивної функції потрібно бути уважним, бо, як і у випадку з циклом while, можна випадково створити функцію з нескінченними викликами. Тому потрібно написати *базовий випадок.*

Головні умови рекурсивних функцій:

- Наявність базового випадка;
- Кроки до базового випадка (можливість його отримання);
- Перший виклик функції не дорівнює базовому випадку

Базовий випадок рекурсивної функції – це кінцевий випадок, коли рекурсивна функція не визиває саму себе.

Прикладом рекурсивної функції для перебору багатомірного масиву і виводу його елементів є:

```
def recurs(mas):
    for i in mas:
        if type(i)==list or type(i)==tuple:
            recurs(i)
        else:
            print(i)

masive_4 = [1, [2,[3,4],5], [6,[7,[8,[9,[10,11],12]]],13], 14]
recurs(masive_4)
```

Вивод:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

Робота з зовнішніми файлами

В мові Python можна створювати, оброблювати та видаляти файли даних. Файли даних можуть бути *текстові* або *бінарними*.

Текстові файли – це файли, які містять дані, що може свободно прочитати людина. У більшості випадків такі файли оброблює програма Блокнот (за стандартом у Windows). Найпоширеніше розширення – txt. Будь-які текстові файли можуть бути перетворені у бінарні.

Бінарні (двійкові) файли – це файли, що складаються з послідовності байтів. Потребують комп'ютерної обробки для розуміння людиною. До таких файлів можуть відноситися абсолютно всі файли компіляції програм. Розширення – bin.

Текстові файли

Відкриття текстового файлу

Відкрити будь-який файл можливо кількома способами. Для цього використовується функція read(). Вона приймає 2 параметри: перший – шлях до файлу, написаний у кавичках; другий – режим відкриття. Функція повертає у програму відкритий файл з подальшою можливістю роботи з ним.

Шлях (з іменем файлу) вказується від папки-директорії, у якій знаходиться програма.

Існують наступні режими відкриття текстового файлу:

- 'w' або 'wt' – режим запис файлу. У цьому режимі можна створити порожній файл, якщо ще не існує файлу зі вказаною назвою, та очистити файл, якщо такий файл існував заздалегідь. При роботі з файлом у цьому режимі можна записувати інформацію в текстовому виді.
- 'x' або 'xt' або 'w+' або 'w+t' – створення нового текстового файлу та запис у нього. Якщо файл з такою назвою існує, то виникає збій.
- 'r' або 'rt' – режим читання файлу. У цьому режимі файл може читатися построково, блоками байтів або цілком. Редагування або створення файлу неможливе.
- 'r+' або 'r+t' – модифікований режим читання файлу. Покажчик читання встановлюється у кінець файлу.
- 'a' або 'at'– режим дозапису. Цей режим дозволяє записати нові дані у кінці вже існуючого текстового файлу.

Якщо режим не вказаний, то автоматично виставляється режим читання r.

```
f2 = open('filezp.txt', 'w')
```

Після завершення роботи з файлом варто його закрити. Це виконується методом .close().

```
f2.close()
```

Альтернативний спосіб відкриття файлу – команда with. Він має заголовок та тіло. У заголовку пишеться команда open() з вказанням шляху та потрібного режиму відкриття та назва змінної, до якої прив'язується файл на час виконання тіла. Вигляд заголовок має наступний:

```
with open('output.txt') as f:
```

В тілі пишуться дії, що виконуються з файлом. Нові записанні змінні у тілі зберігаються для подальшого використання.

```
with open('output.txt') as f:
    data = f.read()
```

Використання команди with дозволяє не закривати файл, так як це робиться автоматично в кінці тіла команди.

Читання текстового файлу

Читання текстового файлу виконується методом *.read()* або *.readline()*.

Метод файлу .read() приймає параметром ціле число – кількість байтів (символів, відповідно), яке буде зчитуватись з файлу. При цьому, наступне читання починається з місця остановки попереднього. Наприклад, якщо файл має 10 символів, в перший раз зчитано 5, то друге зчитування почнеться з 6-го байту. У випадку, якщо число не буде вказано, то файл зчитається цілком.

Метод файлу .readline() зчитує один рядок (до символа *\n*). Кожен наступний виклик методу буде зчитувати наступний рядок.

Запис у текстовий файл

Записати дані у текстовий файл можливі при використанні методів *.write()* або *.writelines()* при відповідному режимі.

Метод .write() приймає строковий тип даних та записує їх у файл. Символ переносу строки – \n (рахується як один символ).

Метод .writelines() приймає список у якості параметру та записує кожен елемент списку у файл.

Бінарні файли, pickle

Створення бінарного файлу, модуль pickle

Читання бінарного файлу людиною неможливе, так як простей ого відкриття у вигляді тексту предоставить незв'язний набор символів. Тому бінарні файли потребують машинної обробки для зручного використання. Для цього використовується модуль pickle.

Відкриття бінарного файлу виконується так само, як і текстові, але режими вказуються інакше:

- 'wb' – режим запису бінарного файлу.
- 'xb' або 'w+b' – створення нового бінарного файлу та запис у нього.
- 'rb' – режим читання двійкового файлу.
- 'r+b' – модифікований режим читання файлу.
- 'ab' – режим дозапису.

Так як звичайне зчитування файлу показує набір символів, то варто перед виводом їх обробити. Для цього використовується функції модуля pickle. Для зчитування за допомогою pickle використовується load(), параметром якого є змінна, прив'язана до відкритого файлу. Функція повертає зчитанні та оброблені дані.

```
>>> with open('data.pickle', 'rb') as f:
...     data_new = pickle.load(f)
```

Для запису у бінарний файл використовується функція dump(). Вона приймає 2 параметри: 1 – інформація, що записується; 2 – файл, куди будуть записуватися дані.

```
>>> with open('data.pickle', 'wb') as f:
...     pickle.dump(data, f)
```

Модуль os

Для роботи з файлами також використовується бібліотека os. Вона не може зчитувати або редагувати зміст файлів, але вона допомагає створювати, переміщувати, видаляти та робити інші дії з файлами.

Існує ряд методів об'єкту path модуля os.

- os.path.isfile() – визначає, чи вказує переданий шлях на файл. Повертає True, якщо переданий шлях до файла і False, якщо до директорії (папки).

```
os.path.isfile(path)
```

- os.path.isdir() – аналогічний до isfile(). Повертає True, якщо вказаний шлях до директорії.

```
os.path.isdir(path)
```

- os.path.exists() – перевіряє, чи існує переданий шлях. Також повертає логічний вираз.

```
os.path.exists(path)
```

Також корисні функції модулю os:

- os.mkdir() – створює нову директорію. Параметром є шлях до цієї директорії з вказанням її назви.

```
os.mkdir(path)
```

- os.rename() – переіменовує шлях до файлу. Приймає 2 параметри: 1 – старий шлях, 2 – новий шлях. Таким чином можна як переіменувати файл, так і перенести його в іншу директорію.

```
os.rename(source, dest)
```

- os.remove() – видаляє файл, шлях якого передається параметром.

```
os.remove(path)
```

- os.rmdir() – видаляє директорію, шлях якої передається параметром.

```
os.rmdir(path)
```

- os.listdir() – повертає список всіх файлів та директорій, які знаходяться за вказаним у параметрі шляхом. Якщо параметр не введений, то виконується для поточної директорії.

```
os.listdir(path)
```

- os.getcwd() – повертає шлях поточної директорії.

```
os.getcwd()
```

- os.stat() – повертає статус файла (вся інформація о файлі), шлях якого переданий у параметрі.

```
os.stat(path)
```

Робота з веб-сторінками

Іноді виникає потреба за допомогою програми відкрити сторонній сайт за посиланням та щось на ньому зробити. Це може бути як автоматичне відкриття сайту при запуску файла, так і функція, яка запускається за конмандою (або подією). Для ціх цілей є стандартний модуль WebBrowser. Він дозволяє відкривати ссилки, обирати браузер або режими відкриття.

WebBrowser

Для початку роботи з модулем потрібно його імпортувати.

```
import webbrowser
```

Найпростішим способом відкриття файлу за допомогою скрипта є функція open(), параметром є посилання. Ця функція відкриває сайт у новій вкладці браузера, встановленного за замовчуванням (стандартно на Windows – InternetExplorer, але в нових версіях – Microsoft Edge).

```
webbrowser.open('http://pythontutor.ru/lessons/')
```

Посилання має бути строкового типу, що надає можливість редагувати їх. Це корисно у випадках, якщо запит введений не при написанні коду, а при вводу посилання користувачєм. Наприклад:

```
link = input('>>> ')
webbrowser.open(link)
```

При цьому варто мати на увазі, що, якщо посилання введено у форматі з https:// або з http://, то сайт відчинеться у браузері за замовчанням (можливо, встановленним користувачєм). У випадку, якщо без, то вказання протоколу, то сайт відчинеться у Internet Explorer.

Можна зробити перевірку на формат посилання. Напишемо функцію:

```
def Valide(link):
    cut_link = link.split('://')
    if cut_link[0] in ('https', 'http'):
        return link
    else:
        return 'http://'+link
```

У випадку, якщо перед ‘://’ стоїть ‘https’ або ‘http’, то функція повертає посилання у тому ж самому вигляді, як і прийняла його. У випадку, якщо якщо перед ‘://’ стоїть не ‘https’ або ‘http’ (або ‘://’ немає взагалі), то функція повертає посилання з доданним до нього ‘http’. У випадку, якщо сайт має протокол https, браузер вставить його автоматично.

За допомогою webbrowser також можливо редагувати режими відкриття веб-сторінки:

- Відкрити у новому вікні браузера за замовчуванням якщо це можливо. У зворотньому випадку відчинить сторінку у єдиному вікні браузера.

```
webbrowser.open_new(link)
```

- URL-адреса відкриється на новій сторінці (“tab”) браузера за замовчуванням, якщо це можливо, інакше еквівалентно open_new().

```
webbrowser.open_new_tab(link)
```

- Відкрити сторінку за допомогою браузера за замовчуванням. Якщо new дорівнює 0, URL-адреса відкривається в тому самому вікні браузера, якщо це можливо. Якщо new дорівнює 1, за можливості відкривається нове вікно браузера. Якщо значення new дорівнює 2, за можливості відкривається нова сторінка браузера (“tab”). Якщо для autoraise значення True, вікно піднімається, якщо це можливо.

```
webbrowser.open(link, new=1, autoraise=True)
```

Припустимо, що вам не потрібен стандартний браузер. Для вибору браузера є команда .get().

```
webbrowser.get(using=None)
```

Грубо кажучи, ви просто вказуєте, який браузер вам використовувати.

Таблиця назв деяких браузерів:

Назва браузера	**Його назва в Python**
Chrome	‘google-chrome’
Chromium	‘chromium-browser’
Opera	‘opera’
Firefox	‘mozilla’
Grail	‘grail’
Safari	‘safari’
MacOS (default browser)	‘macosx’

Також можливо “зареєструвати” браузер, тобто вказати його шлях:

```
webbrowser.register('Chrome', None, webbrowser.BackgroundBrowser(
    'C:\Program Files (x86)\Google\Chrome\Application\chrome.exe'))
```

Після реєстрування браузера всі посилання будуть відкриватися у ньому.

Симуляція управління

В Python є можливість симулювати дію клавіатури за допомогою скриптів. Тобто при запуску скрипта буде надіслано набір символів у вигляді, ніби вони були набраті з клавіатури або мишки користувача. Для цього є зовнішня бібліотека pyautogui.

Для її завантаження (в консолі Windows):

pip install pyautogui

Імпорт бібліотеки:

```
import pyautogui as pag
```

За допомогою PyAutoGUI можливо:

- Дізнатися поточне положення миші на екрані:

  ```
  pag.position()
  ```

  Виход:

  ```
  Point (x = 643, y = 329)
  ```

- Дізнатися висоту та ширину екрана:

  ```
  pag.size()
  ```

  Виход:

  ```
  Size (width = 1440, height = 900)
  ```

- Поставити мишку на задану точку на екрані:

  ```
  pag.moveTo(x_coordinate, y_coordinate)
  ```

- Плавно перемістити мишку на задану кількість пікселів від її поточного положення:

  ```
  pag.moveRel(100, 100, 2)
  ```

  Перший параметр – зсув на 100 пікселів вправо;
  Другий параметр – зсув на 100 пікселів вниз;
  Третій параметр – час зсуву (у секундах).

- Натискання мишкою:

  ```
  pag.click(x, y, clicks, interval, button)
  ```

  X – координата x;
  У – координата y;
  Clicks – кількість натискань у заданній точці;

Interval – час в секундах між натисканнями у точці;
Button – вказання кнопки мищі, якою відбувається натискання (‘right’, ‘left’ або ‘middle’).

- Введення тексту з клавіатури:

```
pag.typewrite(text, interval)
```

Text – строка, що буде введена (також можна передавати масиви строк)
Interval – час в секундах між кожним вводом символа.

- Натискання комбінацій клавіш:

```
pag.hotkey('shift', 'enter')
```

Приймає кілька клавіш, что будуть натиснуті одночасно.

- Скріншот:

```
pag.screenshot('ss.png')
```

Параметр – ім’я, під яким буде збережений скріншот екрана.

Об’єктно орієнтоване програмування (ООП)

Основні поняття ООП

Принцип роботи мови Python складає об’єктовий підхід. Абсолютно все, що вже було вивчено в минулих лекціях, в кореневих файлах мови має об’єктно-класове представлення. Іншими словами, всі діючі одиниці мові Python – це об’єкти.

Поняття ООП

Об’єктно орієнтоване програмування (ООП) – це парадигма програмування, де різні компоненти комп’ютерної програми моделюються на основі реальних об’єктів.

Об’єкт – це одиниця в програмуванні, що має якісь характеристики і те, що може виконати будь-яку функцію.

Клас в об’єктно-орієнтованом програмуванні виступає в ролі креслення для об’єкта. Клас має опис властивостей, методів та функцій об’єкта, але сам їм не являється.

Об’єкт та клас – це реально існуючі одиниці в коді, а не лише поняття. Об’єкти народжуються від своїх класів. В мові програмування Python такі об’єкти зазвичай називають екземплярами.

Головні принципи ООП

ООП має 3 основних принципи: спадкування, поліморфізм та інкапсуляція.

Спадкування – це умовне поняття, яке передбачає собою передавання властивостей від батьків до дочірніх елементів. У данному випадку – від класу до екземпляру або від старшого класу до молодшого. Властивість, яку має батьківський клас, буде мати і дочірній, і їх екземпляри.

Поліморфізм – це безліч форм. Тобто, екземпляри різних класів з різною внутрішньою складовою можуть мати однакові інтерфейси. Наприклад, інтерфейс “+” має як і клас чисел, так і клас рядків, але, якщо підклас int з підкласом float буде складувати числа, то клас str конкантенує свої об’єкти (рядки).

Інкапсуляція – це приховування даних, тобто відсутність можливості отримати їх напряму. Для отримання існують інтерфейси. У Python інкапсуляція відсутня, але, при необхідності, її можна імітувати.

Створення об’єктів та класів

Створення класу

В мові програмування Python класи створюються дуже легко: пишемо class, його назву та вписуємо його властивості, які будуть передаватися об’єктам.

```
class A():
    властивості
```

Якщо клас є дочірним (тобто він сам буде належати якомусь класу), то батьківський клас указується у скобках.

Щоб написати властивістьдля класу, потрібно створити поле.

Поле класу (атрибут) – це змінна, зв'язана з класом або об'єктом. Всі дані об'єкта зберігаються в його полях. Доступ до полів здійснюється по імені. Зазвичай тип даних кожного поля задається в описі класу, членом якого є поле.

```
class A():
    m = 20
    l = 10
```

При цьому наслідування буде передавати ці поля дочірнім класам та об’єктам створенного класу.

Створення об’єкту

Створення об’єкту відбувається шляхом простого виклика класу. При цьому скобки писати обов’язково. Також об’єкту потрібно присвоїти змінну.

```
n = A()
```

Для окремого об’єкту ми можемо змінити або додати властивість, тобто переписати поле.

```
n = A()
n.l = 30
n.k = 50
```

Однак, у програмуванні майже не використовується додавання властивості для окремого об’єкту, так як у великому проекті різні властивості будуть вносити хаос в коді.

Створення функцій класу

Для додання функції класу пишемо функцію у тілі класу. При написанні функції обов'язковим першим параметром є змінна, яка позначає об'єкт.

```
class A():
    m = 20
    l = 10
    def dodavannya(self):
        return self.m+3
```

За внегласним правилом зазвичай це є self, але ця змінна може мати будь-яку назву. При цьому у дужках при виклику функції ми не вказуємо об'єкт. Це робиться тому, що функція належить класу. При читанні запису виклику функції інтерпретатор шукає функцію у властивостях об'єкту, не знаходить та шукає її у властивостях класу цього об'єкта. Коли він знаходить функцію, запис, незримо для нас, перетворюється на виклик функції класу з вказаним об'єктом в якості першого параметру. Всі інші вказані параметри зміщуються на наступну позицію. Функції класу також називаються його методами.

Будьте уважні, що поля класу і поля об'єктів – це різні поля.

Атрибут __dict__

В мові програмування Python є атрибут об'єктів __dict__. При його виклику він повертає словник з властивостями об'єкту у ключах та їх значеннями у значеннях словника.

```
print(n.__dict__)
```

```
{'l': 30, 'k': 50}
```

Але при застосуванні атрибута до об'єкта у нього додаються тільки змінені або додані властивості, так як всі інші належать саме класу.

Атрибут __dict__ можна застосовувати до класів, але при цьому запис також повертає властивості, які записані за замовчуванням:

```
print(A.__dict__)
```

```
{'__module__': '__main__', 'm': 20, 'l': 10, 'dodavannya': <function A.dodavanny
a at 0x000001E7B337F3A0>, '__dict__': <attribute '__dict__' of 'A' objects>, '__
weakref__': <attribute '__weakref__' of 'A' objects>, '__doc__': None}
```

Атрибути класів

Якщо викликати список атрибутів класа, то можна побачити наступне:

[‘__class__’, ‘__delattr__’, ‘__dict__’, ‘__dir__’, ‘__doc__’, ‘__eq__’, ‘__format__’, ‘__ge__’, ‘__getattribute__’, ‘__gt__’, ‘__hash__’, ‘__init__’, ‘__init_subclass__’, ‘__le__’, ‘__lt__’, ‘__module__’, ‘__ne__’, ‘__new__’, ‘__reduce__’, ‘__reduce_ex__’, ‘__repr__’, ‘__setattr__’, ‘__sizeof__’, ‘__str__’, ‘__subclasshook__’, ‘__weakref__’, ‘make’, ‘model’, ‘name’, ‘start’, ‘stop’]

Атрибути можуть бути віднесені до *атрибутів класа* та *атрибутів екземпляра.*

Атрибути класу є власністю класа, *атрибути екземплярів* є власністю екземпляра. *Атрибути екземпляра* оголошуються всередині будь-якого методу (функції класу), тоді як *49трибутик ласу* оголошуються поза будь-яким методом.

Одним з найбільше використовуваних атрибутів є __init__. Це є конструктором полів екземпляру. Він записується аналогічно функції класа, але спрацьовує при створенні нового об’єкту.

```
class Person:
    def set_name(self, n, s):
        self.name = n
        self.surname = s
```

Таким чином можна встановити окремі поля одразу при створенні нового об’єкту. У такому випадку треба створювати екземпляри з вказанням значень полів у якості параметрів.

```
>>> p1 = Person()
>>> p1.set_name("Bill", "Ross")
>>> p1.name, p1.surname
('Bill', 'Ross')
```

Можна встановити параметри по замовчуванню. Для цього можна одразу встановити значення полів.

```
class Rectangle:
    def __init__(self, w=0.5, h=1):
        self.width = w
        self.height = h
```

Також можна за допомогою __init__ передати поля класу до об’єкта.

Також існує атрибут __del__. Він виконується при видаленні об’єкта. Немає сенсу вводити туди нові поля, але є сенс змінювати поля класа, виводити написи або вертати повідомлення.

__call__ – автоматично викликається при зверненні до об’єкта як до функції.

Також є атрибути перегрузки. *Перегрузка функції* – це заміна дії інтерфейсу для окремого класу або групи класів.

Приклади декількох атрибутів перегрузки:

__add__ – застосовується для заміни інтерфейса додавання (“+”);
__sub__ – заміна інтерфейса віднімання (“-“);
__mul__ – заміна множення (“*”);
__truediv__ – заміна ділення.

Також є наступні атрибути об’єктів:

- __class__ – вертає назву класу об’єкта.
- __doc__ – вертає документацію класу.
- __getattribute__ – вертає назву атрибуту у вигляді рядка.
- __str__ – вертає об’єкт у строковому вигляді.

Принципи ООП

ООП, не залежно від мови програмування, має 3 основних принципи: *спадкування, поліморфізм* та *інкапсуляція.* У мові Python чітко викреслені тільки спадкування та поліморфізм. Інкапсуляція є скоріш умовною, ніж явною.

Як вже було сказано, робота Python базується на об'єктно-орієнтованому програмуванні, тому для всіх класів (в тому числі вбудованих, таких як int, float або str) виконуються ці принципи.

Спадкування

Спадкування – це передача полів та функцій від батьківських класів дочірнім і, відповідно, їх об'єктам.

Просте спадкування – це передача методів (функцій) батьківського класу до одного підкласу. *Множинне спадкування* відбувається так само, але для кількох підкласів.

```
class Table:
    def __init__(self, l, w, h):
        self.length = l
        self.width = w
        self.height = h


class DeskTable(Table):
    def square(self):
        return self.width * self.length


t1 = Table(1.5, 1.8, 0.75)
t2 = DeskTable(0.8, 0.6, 0.7)
```

```
class Table:
    def __init__(self, l, w, h):
        self.length = l
        self.width = w
        self.height = h


class DeskTable(Table):
    def square(self):
        return self.width * self.length


class ComputerTable(DeskTable):
    def square(self, monitor=0.0):
        return self.width * self.length - monitor
```

При простому та множинному спадкуванні до дочірнього класу переходять як і атрибути, так і функції батьківського. Окрім устаткованих, дочірній клас може мати свої параметри, які працюють у цьому класі та в його дочірних. Також один клас може мати декілька батьківських. У такому разі підклас буде устатковувати параметри від усіх батьків.

Поліморфізм

Поліморфізм проявляється в перегрузці методів або його перевизначення. Перегрузка метода (функції) – це властивість метода вести себе по-різному в залежності від кількості або типа параметрів.

```
class Car:
  def start(self, a, b=None):
    if b is not None:
      print (a + b)
```

```
else:
    print (a)
```

Перевизначення метода (функції) – це властивість методів з однаковою назвою, що відносяться до батьківського та дочірнього класу. Визначення метода відрізняється в батьківському та дочірньому класах, але назва залишається тією ж. У такому разі у дочірньому класі власний метод перекриває унаслідований від батьків.

```
class Vehicle:
  def print_details(self):
    print("Це батьківський метод класа класса Vehicle")

class Car(Vehicle):
  def print_details(self):
    print("Це дочірній метод класа Car")

class Cycle(Vehicle):
  def print_details(self):
    print("Це дочірній метод класа Cycle")
```

Інкапсуляція

Інкапсуляція просто означає приховування даних. Як правило, в об'єктно-орієнтованому програмуванні один клас не повинен мати прямого доступу до даних іншого класу. Натомість, доступ повинен контролюватись через методи класу. Для контролю використовуються модифікатори доступу.

Модифікатори доступу – це позначки змінних, які визначають область видимості цієї змінної. Модифікаторів доступу в Python існує 3 вида:

- Публічний (public) – доступ до змінної відкритий з будь-якого місця поза класом.
- Приватний (private) – доступ відкритий тільки у межах класу.
- Захищений (protected) – доступ відкритий тільки у межах одного пакету.

За замовчуванням створюються атрибути з публічним модифікатором доступу. Для створення приватного модифікатора перед назвою змінної ставлять префікс __, для створення захищенного модифікатора – префікс _.

Змінні з однаковою назвою, але з різними модифікаторами є різними змінними.

Доступ до значення змінної з приватним модифікатором доступу можливо отримати тільки через метод класу.

Фреймовки, Tkinter

Поняття фреймворку

Фреймворк, відмінності від бібліотек

Фреймворк – це бібліотека, яка повністю визначає структуру програми; програмне забеспечення, визначаюче структуру програмної системи; програмне забеспечення, яке значно полегшує роботу програміста та дозволяє об'єднати різні компоненти проекта. Фреймворк також може використовувати велику кількість бібліотек для власної роботи.

Відмінність від бібліотеки та, що при імпорті бібліотеки встановлюється набор додаткових функцій для користування в файлі програми. При імпорті фреймворку змінюється архітектура програми. В деяких випадках створюється набір папок, необхідних для проекта.

Прикладами крупних фреймворків на мові Python є:

- Django – фреймворк для ефективного створення сайтів та веб-додатків. До його сильних сторін належать такі можливості:
  - Аутентифікація
  - Маршрутизація URL-адрес
  - Робота з базами даних: PostgreSQL, MySQL, SQLite, Oracle
  - Підтримка веб-серверів та інш.
- CherryPy – мікрофреймворк для розробки додатків для Android. Є частиною фреймворку TurboGears. Має готові інструменти для аутентифікації, кешування, кодування, статичного контенту.
- TurboGears – фреймворк для створення веб-додатків, які працюють з даними. Вважається альтернативою Django.
- Pyramid – фреймворк, який має великий набір функцій. Є каркасом для як малих, так і великих додатків.
- Bottle – фреймворк для роботи над малими та середніми проектами. При цьому код фреймворка складається лише з одного файла.
- Tkinter – стандартний фреймворк для створення графічного інтерфейсу користувача. Може бути встановлений разом з інтепретатором мови Python, тому часто не потребує попереднього встановлення. Фреймворк реалізований на мові Tcl, але має адаптацію для роботи з Python.

Tkinter

Tkinter – це фреймворк для Python, призначений для роботи з бібліотеками Tk. Бібліотеки Tk містить компоненти графічного інтерфейса користувача, написані на мові програмування Tcl.

Графічний інтерфейс користувача (GUI) — це тип інтерфейсу, який дає змогу користувачам взаємодіяти з електронними пристроями через графічні зображення та візуальні вказівки, на відміну від текстових інтерфейсів, заснованих на використанні тексту, текстовому наборі команд та текстовій навігації. Виконання дій у GUI— це безпосередня маніпуляція з графічними елементами.

Tkinter імпортується так само, як і інші бібліотеки.

```
import tkinter as tk
```

Далі, щоб створити програму з GUI, потрібно виконати такі дії:

1) Створити головне вікно.

2) Створити віджети (елементи) цього вікна.

3) Приєднати до елементів властивості (функції).

4) Вибрати події, на які буде реагувати програма.

5) Розташувати елементи на вікні.

6) Запустити цикл обробки подій.

Для створення вікна потрібно створити об’єкт класу Tk.

```
win = tk.Tk()
```

Створення елементів відбувається після створення головного вікна. Всі вони є об’єктами. Від класу залежить тип елемента, а від переданих при виклику параметрів – зовнішній вигляд та функціонал віджета.

Для створення однострочного поля з текстом використовується клас Label. Його параметрами можуть бути:

- Назва вікна, у яке буде поміщений віджет. У разі, якщо вікно не вказано, віджет буде поміщенний у головне вікно. Цей параметр завжди передається на першому місці.
- Width – довжина строки (у символах).
- Text – строка всередені віджета.
- Font – шрифт та розмір текста
- Bg – колір фона віджета
- Fg – колір напису
- Justify – положення тексту в віджеті

Також є багато інших параметрів, які регулюють больш тонке налаштування.

```
lab = tk.Label(win, width='15', bg='#0bb1bd', text='Приклад тексту', justify='center')
```

За допомогою функцій строкове поле може отримувати нові або змінювати старі поля.

Щоб віджет з’явився у вікні, потрібно йог отуди помістити. Найпростішим методом для цього є метод pack(). Він належить усім віджетам. При його використанні без додаткових параметрів, розміщення віджетів буде послідовним у стовпець на верхній частині вікна.

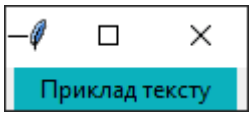
```
lab.pack()
```

Останній крок – зациклення вікна. Цей крок обов’язковий, так як без нього вікно буде закриватися одразу ж після першого кадру. Для цього використовується метод mainloop(). Цей метод доступний лише вікнам.

```
win.mainloop()
```

Як результат ми маємо такий код і результат виконання:

```
1  import tkinter as tk
2
3  win = tk.Tk()
4  lab = tk.Label(win, width='15', bg='#0bb1bd',
5                 text='Приклад тексту', justify='center')
6
7  lab.pack()
8
9  win.mainloop()
```



Віджети Label, Entry, Button

Головне вікно

Виклик головного вікна не має додаткових параметрів для налаштування зовнішнього вигляда, але має ряд методів, які дозволяють змінити вигляд вікна. До них відносяться зміна розміру вікна, його відстань від країв монітору, колір фона та інш.

- .geometry() – зміна розміру вікна.

Його параметр – це строка типу ‘ширина х висота + відступ зверху + відступ зліва’.

```
win = tk.Tk()
win.geometry('300x300+200+100')
```

Якщо висота та ширина екрана невідома, то варто додати дві нові змінні:

```
w = win.winfo_screenwidth() #Визначаємо ширину екрана
h = win.winfo_screenheight() #Визначаємо висоту екрана
```

Це є число. Ми можемо його поділити або відняти якусь кількість пікселів. Додавати або множити це число не варто, бо тоді вікно вийде за границю екрану.

Далі ці змінні ми вставляємо до методу розмірів:

```
win.geometry('%dx%d+0+0' %(w,h))
```

- .title() – зміна заголовку вікна.

```
win.title('Моя програма')
```

- .overrideredirect() – прибирає строку назви вікна.
- .destroy() – закриває вікно, може використовуватися в функціях.
- .configure() – параметри вікна.

До параметрів цього метода можуть відноситися background. Він змінює колір фона всього вікна.

```
win.configure(background='pink')
```

Також є багато інших параметрів.

- .resizable() – дозволяє або забороняє розширення вікна.

Віджети label, entry, button

Створення елементів відбувається після створення головного вікна. Всі вони створюються методом вікна з прив’язкою до змінної. Наприклад:

```
lab = tk.Label(width='50', text='Введіть текст') #Створюємо напис
ent = tk.Entry(width=50) #Поле для вводу
but = tk.Button(text='Переробити') #Кнопку
rez = tk.Label(width='50', text='') #Та пусте поле для напису
```

Віджети зачасту мають такі ж параметри, як і вікно.

Поле Entry має кілька власних методів:

- .get() – отримати введений текст. У випадку, якщо аргументи не приведені, то метод зчитує весь текст. Якщо приведені 2 числових аргументи, то метод зчитує зріз з введеної строки.
- .delete() – видаляє введений текст. Аргументи аналогічні до методу get.
- .insert() – вставляє строку в поле Entry. Перший аргумент – позиція (індекс) вставлення строки (якщо текст повинен бути в кінці – END), другий аргумент – строка, що вставляється.

Для додання функціоналу віджетів треба додати функцію та прив’язати її до події.

Для редагування якоїсь властивості елементу вказується сам елемент, назва властивості у квадратних дужках та цьому виразу присвоюється значення.

Наприклад, якщо при натисканні ми хочемо зчитувати строку з поля вводу та виводити її у поле виводу тексту, то треба виконати наступне:

```
def vyvid(event):
    text = ent.get()
    rez['text'] = str(text)
```

Властивість елемента обов’язково повинна бути строкового типу.

Для того, щоб указати до чого прив’язана функція та при якій події вона виконується, використовують метод *bind()*. У всередені дужок ми пишемо подію та функцію, яка виконується. Наприклад:

```
but.bind('<Button-1>', vyvid)
```

У данному випадку ми прив’язуємо виконання функції до натискання лівою кнопкою миші на кнопку.

Основні події в Python:

<Button-1> – натискання лівої кнопки миші.

<Button-2> – настикання середньої кнопки миші (коліщатко).

<Button-3> – натискання правої кнопки миші.

<Enter> – введеня курсору на зону елемента.

<Leave> – виведення курсору з зони елемента.

<Key> – натискання на будь-яку клавішу.

<Return> – натискання на Enter.

<BackSpace> – натискання пробілу.

<Shift_L> – натискання будь-якого Shift.

<Control_L> – натискання будь-якого Ctrl.

Також у «скобках» < > можна написати будь-яку клавішу – на цю клавішу і буде прив’язана функція. Звертайте увагу на регістр: <r> та <R> – це різні клавіши. Найбільш ефективно це прив’явузвати до полей вводу тексту.

Текстове поле text, scrollbar, метод pack

Текстове поле text

В Tkinter за допомогою віджета text створюється багатострокове текстове поле. За замовчуванням текстове поле за розміром 80 на 24 символи. Для зміни розмірів використовуються параметри width і height. Також є можливість налаштовувати колір фону, тексту, шрифт та інш.

Параметр wrap налаштовує перенос слів на іншу строку. Наприклад, значення WORD буде переносити слово в нову строку цілком.

```
text = Text(width=50, height=30, wrap=WORD)
```

У текстового поля text є власні методи. Вони дозволяють працювати з текстом. До них належать:

- .get() – дозволяє взяти введений в поле текст. Потребує вказання початкового і кінцевого символу (аналогічно до взяття зрізу масиву), але в кожному з аргументів вказується номер строки та номер символа через крапку. При цьому відлік символів починається з 0, а відлік строк – з 1. Якщо потрібно взяти частину тексту з якогось моменту до кінця, то другим аргументом вказується END.

```
string = text.get(1.0, END)
```

- .delete() – видаляє частину тексту. Аргументи розраховуються і вказуються так само, як і в методі get.

```
text.delete(1.0, END)
```

- .insert() – дозволяє вставити частину тексту в текстове поле. Першим аргументом є індекс символу, в який буде вставлятися строка, другим – сама строка.

```
text.insert(1.0, string)
```

Scrollbar

Якщо в текстове поле вводиться більше ліній тексту, ніж його висота, воно саме прокручуватися вниз. При перегляді прокручувати вгору-вниз можна за допомогою коліщатка миші або стрілками на клавіатурі. Проте буває зручніше користуватися скролером - смугою прокручування.

У tkinter скролери виробляються від класу Scrollbar. Об'єкт-скроллер пов'язують із віджетом, якому він потрібний.

```
scroll = Scrollbar(command=text.yview)
```

Створюється скроллер, якого з допомогою опції command прив'язується прокрутка текстового поля по осі y – text.yview.

Для того, щоб прив’язати віджет до скроллера, потрібно ввести його конфігурацію. Для цього в метод віджета .config() треба ввести параметр yscrollcommand зі значенням встановлення потрібного скроллера.

```
text.config(yscrollcommand=scroll.set)
```

Метод pack

Для розміщення віджетів у вікні існує кілька пакувальників: packer, grid, place. Найбільш простий для розуміння та операцій – це packer. Він викликається методом віджета .pack().

Якщо в пакувальники не передавати аргументи, то віджети будуть розміщуватися вертикально, один над одним. Той об'єкт, який першим викличе пакет, буде вгорі. Який другим – під першим, і таке інше.

Метод pack має параметр side (сторона), який приймає одне з чотирьох значень-констант tkinter – TOP, BOTTOM, LEFT, RIGHT (верх, низ, ліво, право). За замовчуванням, коли pack не вказується side, його значення дорівнює TOP.

```
text.pack(side=LEFT)
```

Якщо потрібно розмістити елементи квадратом, то застосовують рамку Frame, тому що комбінування розташувань віджетів на різних сторонах зазвичай не дає такого результату. Фрейми розміщують на головному вікні, а вже у фреймах – віджети. Також існує рамка з написом – об’єкт класу LabelFrame. Вона має такі ж властивості, як і звичайний Frame та властивість text, яка задає підпис рамки.

```
frame = LabelFrame(win, text='Сектор')
```

```
text = Text(frame, width=50, height=30, wrap=WORD)

scroll = Scrollbar(frame, command=text.yview)
```

```
frame.pack(side=BOTTOM)
text.pack(side=LEFT)
scroll.pack(side=LEFT)
```

Також у pack є методи, які дозволяють створити внутрішні (ipadx та ipady) та зовнішні (padx та pady) відступи.

Наступні дві властивості – fill (заповнення) та expand (розширення). За замовчуванням expand дорівнює нулю (інше значення – одиниця), а fill – NONE (інші значення BOTH, X, Y).

Якщо почати розширювати вікно або одразу розкрити його на весь екран, то мітка виявиться вгорі по вертикалі та в середині по горизонталі. Якщо встановити властивість expand в 1, то при розширенні вікна мітка завжди буде в середині.

```
frame.pack(side=BOTTOM, expand=1)
```

Властивість fill змушує віджет заповнювати весь доступний простір. Заповнити його можна у всіх напрямках або лише по одній з осей.

```
scroll.pack(fill=Y, side=LEFT)
```

Остання опція методу pack – anchor (якір) – може набувати значення N (north – північ), S (south – південь), W (west – захід), E (east – схід) та їх комбінації.

Список літератури

Інтернет-література:

1. https://younglinux.info/python.php
2. https://python-scripts.com/how-to-make-an-iterator-in-python
3. https://python-scripts.com/decorators-with-arguments
4. https://python-scripts.com/tkinter
5. https://python-scripts.com/upgrade-pip-windows
6. https://stackoverflow.com/users/17740236/
7. https://pythobyte.com/getting-started-with-python-pyautogui-adc2a5ae/
8. https://habr.com/ru/post/470938/
9. https://www.delftstack.com/ru/howto/python/python-text-to-speech/
10. https://smartiqa.ru/blog/python-list-comprehension

Книжкова література:

1. “Від моноліту до створення мікросервісів”, автор Sem Newell, редакція O-Reilly, 2016 р.
2. “A byte of Python”, автор Swaroop C.H., 2013 р.
3. “Грокаємо алгоритми”, автор Адітья Бхаргава, 2017 р.
4. Електронний підручник “Верстка”, автор Трепачов Д.В., 2012-2022 р.
5. Електронний підручник “JavaScript”, автор Трепачов Д.В., 2012-2022 р.
6. “Програміст фанатік”, автор Chad Fowler, редакція Пітер, 2015 р