Зміст

- Умови
- Python Match
- Цикл While
- Цикл For
- Функції
- Масиви

Умови та оператори `if` у Python

Python підтримує звичайні логічні умови з математики:

- Рівність: `a == b`
- Нерівність: `a != b`
- Менше: `a < b`
- Менше або дорівнює: `a <= b`
- Більше: `a > b`
- Більше або дорівнює: `a >= b`

Ці умови можна використовувати різними способами, найчастіше в операторах `if` та циклах.

Оператор `if` записується за допомогою ключового слова `if`.

Приклад

```
a = 33
b = 200
if b > a:
    print("b більше за a")
```

У цьому прикладі ми використовуємо дві змінні, `a` та `b`, які є частиною оператора `if`, щоб перевірити, чи `b` більше за `a`. Оскільки `a` дорівнює 33, а `b` дорівнює 200, ми знаємо, що 200 більше за 33, тому виводимо на екран "b більше за a".

Відступи

Python використовує відступи (пробіли на початку рядка) для визначення області видимості коду. Інші мови програмування часто використовують фігурні дужки для цієї мети.

Приклад

Оператор `if` без відступів (викличе помилку):

```
```

```
a = 33
b = 200
if b > a:
print("b більше за a")  # Ви отримаєте помилку
```

---

elif

Ключове слово `elif` в Python означає "якщо попередні умови не були істинними, спробуй цю умову".

Приклад

```
a = 33
b = 33
if b > a:
    print("b більше за a")
elif a == b:
    print("a і b рівні")
```

У цьому прикладі `a` дорівнює `b`, тому перша умова не є істинною, але умова `elif` є істинною, тому ми виводимо на екран "a і b рівні".

---

else

Ключове слово `else` обробляє все, що не було враховано попередніми умовами.

Приклад

```
a = 200
b = 33
if b > a:
    print("b більше за a")
elif a == b:
    print("a і b рівні")
else:
    print("a більше за b")
```

У цьому прикладі `a` більше за `b`, тому перша умова не є істинною, також умова `elif` не є істинною, тому ми переходимо до умови `else` і виводимо на екран "a більше за b".

Скорочений запис `if`

Якщо у вас є лише одна команда для виконання, її можна записати в одному рядку з оператором `if`.

Приклад

```
if a > b: print("a більше за b")
```

Скорочений запис `if ... else`

Якщо у вас є лише одна команда для виконання для `if` і одна для `else`, їх можна записати в одному рядку.

Приклад

```
a = 2
b = 330
print("A") if a > b else print("B")
```

Ця техніка відома як тернарний оператор або умовний вираз.

Приклад з кількома умовами

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

`and`

Ключове слово `and` є логічним оператором і використовується для об'єднання умов.

Приклад

```
a = 200
b = 33
c = 500
if a > b and c > a:
    print("Обидві умови істинні")
```

`or`

Ключове слово `or` є логічним оператором і використовується для об'єднання умов.

Приклад

```
a = 200
b = 33
c = 500
if a > b or a > c:
    print("Принаймні одна з умов істинна")
```

---

not

Ключове слово not є логічним оператором і використовується для інверсії результату умови.

Приклад

```
a = 33
b = 200
if not a > b:
    print("a НЕ більше за b")
```

Вкладені оператори if

Ви можете використовувати оператори if всередині інших операторів if. Це називається вкладеними умовами.

Приклад

```
x = 41

if x > 10:
    print("Більше за 10,")
    if x > 20:
        print("і також більше за 20!")
    else:
        print("але не більше за 20.")
```

Оператор pass

Оператори if не можуть бути порожніми, але якщо з якоїсь причини вам потрібен порожній оператор if, використовуйте оператор pass, щоб уникнути помилки.

Приклад

```

```

```
a = 33
b = 200

if b > a:
    pass
```

Python Match

Оператор `match` використовується для виконання різних дій залежно від умов.

Оператор `match` у Python

Замість написання багатьох операторів `if..else`, ви можете використовувати оператор `match`.

Оператор `match` обирає один із багатьох блоків коду для виконання.

Синтаксис

```
match expression:
    case x:
        # блок коду
    case y:
        # блок коду
    case z:
        # блок коду
```

Як це працює:

1. Вираз `match expression` обчислюється один раз.
2. Значення виразу порівнюється зі значеннями кожного `case`.
3. Якщо є збіг, виконується відповідний блок коду.

Приклад

Використання номера дня тижня для виведення назви дня:

```
day = 4
match day:
    case 1:
        print("Понеділок")
    case 2:
        print("Вівторок")
    case 3:
        print("Середа")
    case 4:
        print("Четвер")
```

```
    case 5:
        print("П'ятниця")
    case 6:
        print("Субота")
    case 7:
        print("Неділя")
```

Значення за замовчуванням

Використовуйте символ підкреслення _ як останнє значення case, якщо потрібно виконати блок коду, коли немає інших збігів:

Приклад

```
day = 4
match day:
    case 6:
        print("Сьогодні субота")
    case 7:
        print("Сьогодні неділя")
    case _:
        print("Чекаю на вихідні")
```

Значення _ завжди збігається, тому важливо розміщувати його останнім, щоб воно працювало як значення за замовчуванням.

Комбінування значень

Використовуйте символ | як оператор "або" у виразі case, щоб перевірити збіг із кількома значеннями в одному case:

Приклад

```
day = 4
match day:
    case 1 | 2 | 3 | 4 | 5:
        print("Сьогодні робочий день")
    case 6 | 7:
        print("Я люблю вихідні!")
```

Умови в case

Ви можете додати оператори if у виразі case для додаткової перевірки умов:

Приклад

```
month = 5
day = 4
match day:
    case 1 | 2 | 3 | 4 | 5 if month == 4:
        print("Робочий день у квітні")
    case 1 | 2 | 3 | 4 | 5 if month == 5:
        print("Робочий день у травні")
    case _:
        print("Немає збігу")
```

Цикли Python While

Цикли Python

Python має дві примітивні команди циклу:

- цикли `while`
- цикли `for`

Цикл `while`

За допомогою циклу `while` ми можемо виконувати набір операторів, якщо умова виконується.

Приклад: Вивести `i`, якщо `i` менше 6:

```
i = 1
while i < 6:
    print(i)
    i += 1
```

> **Примітка:** Не забудьте збільшити `i`, інакше цикл триватиме вічно.

Цикл `while` вимагає, щоб відповідні змінні були готові. У цьому прикладі нам потрібно визначити змінну індексування `i`, яку ми встановили на `1`.

Заява про `break`

За допомогою оператора `break` ми можемо зупинити цикл, навіть якщо умова `while` виконується.

Приклад: Вийти з циклу, коли `i` буде 3:

```
i = 1
while i < 6:
```

```
    print(i)
    if i == 3:
        break
    i += 1
```

Заява про `continue`

За допомогою оператора `continue` ми можемо зупинити поточну ітерацію та продовжити наступну.

Приклад: Перейдіть до наступної ітерації, якщо `i` дорівнює 3:

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

Інструкція `else`

За допомогою оператора `else` ми можемо запустити блок коду один раз, коли умова більше не виконується.

Приклад: Надрукуйте повідомлення, коли умова не виконується:

```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

Python для циклів

Цикл `for`

Цикл `for` використовується для повторення послідовності (тобто списку, кортежу, словника, набору або рядка).

Це менше схоже на ключове слово `for` в інших мовах програмування, а працює більше як метод ітератора, який є в інших мовах об'єктно-орієнтованого програмування.

За допомогою циклу `for` ми можемо виконати набір операторів один раз для кожного елемента в списку, кортежі, наборі тощо.

Приклад: Надрукуйте кожен фрукт у списку фруктів:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

> **Примітка:** Цикл `for` не вимагає попередньої установки змінної індексування.

Цикл через рядок

Навіть рядки є ітерованими об'єктами, вони містять послідовність символів.

Приклад: Переберіть букви в слові «банан»:

```
for x in "banana":
    print(x)
```

Заява про `break`

За допомогою оператора `break` ми можемо зупинити цикл до того, як він пройде через усі елементи.

Приклад: Вийти з циклу, коли `x` є "банан":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
    if x == "banana":
        break
```

Приклад: Вийти з циклу, коли `x` є "банан", але цього разу перерва буде перед друком:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        break
    print(x)
```

Заява про `continue`

За допомогою оператора `continue` ми можемо зупинити поточну ітерацію циклу та продовжити наступну.

Приклад: Не друкувати "банан":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

Функція `range()`

Щоб прокрутити набір коду задану кількість разів, ми можемо використати функцію `range()`.

Функція `range()` повертає послідовність чисел, починаючи з `0` за замовчуванням і збільшуючи на `1` (за замовчуванням), і закінчується вказаним числом.

Приклад: Використання функції `range()`:

```
for x in range(6):
    print(x)
```

> **Примітка:** `range(6)` – це не значення від `0` до `6`, а значення від `0` до `5`.

Функція `range()` за замовчуванням має значення `0` як початкове значення, однак можна вказати початкове значення, додавши параметр: `range(2, 6)`, що означає значення від `2` до `6` (але не включаючи `6`).

Приклад: Використання параметра `start`:

```
for x in range(2, 6):
    print(x)
```

Функція `range()` за замовчуванням збільшує послідовність на `1`, однак можна вказати значення збільшення, додавши третій параметр: `range(2, 30, 3)`.

Приклад: Збільште послідовність на `3` (за замовчуванням `1`):

```
for x in range(2, 30, 3):
    print(x)
```

Інше в циклі `for`

Ключове слово `else` в циклі `for` визначає блок коду, який буде виконано після завершення циклу.

Приклад: Вивести всі числа від `0` до `5` і надрукувати повідомлення, коли цикл завершиться:

```
for x in range(6):
    print(x)
else:
    print("Finally finished!")
```

> **Примітка:** Блок `else` **не** буде виконано, якщо цикл зупинено оператором `break`.

Приклад: Розірвіть цикл, коли `x` буде `3`, і подивіться, що станеться з блоком `else`:

```
for x in range(6):
    if x == 3:
        break
    print(x)
else:
    print("Finally finished!")
```

Вкладені цикли

Вкладений цикл — це цикл усередині циклу.

«Внутрішній цикл» буде виконуватися один раз для кожної ітерації «зовнішнього циклу».

Приклад: Виведіть кожен прикметник для кожного фрукта:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
    for y in fruits:
        print(x, y)
```

Заява про `pass`

Цикли `for` не можуть бути порожніми, але якщо у вас з якоїсь причини є цикл `for` без вмісту, вставте оператор `pass`, щоб уникнути помилки.

Приклад:

```
for x in [0, 1, 2]:
    pass
```

Функції в Python

Функція — це блок коду, який виконується лише тоді, коли його викликають.

Ви можете передавати дані у функцію, які називаються параметрами.

Функція може повертати дані як результат.

Створення функції

У Python функція визначається за допомогою ключового слова `def`:

```
def my_function():
    print("Привіт з функції")
```

Виклик функції

Щоб викликати функцію, використовуйте її ім'я, за яким слідують круглі дужки:

```
def my_function():
    print("Привіт з функції")

my_function()
```

Аргументи

Інформацію можна передавати у функції як аргументи. Аргументи вказуються після імені функції в круглих дужках. Ви можете додати стільки аргументів, скільки потрібно, розділяючи їх комами.

```
def my_function(fname):
    print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Параметри чи аргументи?

- **Параметр** — це змінна, вказана в дужках під час визначення функції.
- **Аргумент** — це значення, яке передається функції під час її виклику.

Кількість аргументів

За замовчуванням функцію потрібно викликати з правильною кількістю аргументів:

```
def my_function(fname, lname):
    print(fname + " " + lname)
```

```
my_function("Emil", "Refsnes")
```

Якщо кількість аргументів неправильна, виникне помилка.

Довільні аргументи (*args)

Якщо ви не знаєте, скільки аргументів буде передано, додайте * перед іменем параметра:

```
def my_function(*kids):
    print("Наймолодша дитина — " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

Іменовані аргументи (kwargs)

Ви можете передавати аргументи у вигляді ключ = значення. Порядок аргументів у цьому випадку не має значення:

```
def my_function(child3, child2, child1):
    print("Наймолодша дитина — " + child3)

my_function(child1="Emil", child2="Tobias", child3="Linus")
```

Довільні іменовані аргументи (**kwargs)

Якщо ви не знаєте, скільки іменованих аргументів буде передано, додайте ** перед іменем параметра:

```
def my_function(**kid):
    print("Його прізвище — " + kid["lname"])

my_function(fname="Tobias", lname="Refsnes")
```

Значення параметра за замовчуванням

Ви можете встановити значення параметра за замовчуванням:

```
def my_function(country="Norway"):
    print("Я з " + country)

my_function("Sweden")
my_function("India")
```

```
my_function()
my_function("Brazil")
```

Передача списку як аргументу

Ви можете передати будь-який тип даних у функцію. Наприклад, список:

```
def my_function(food):
    for x in food:
        print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Повернення значень

Щоб функція повертала значення, використовуйте оператор `return`:

```
def my_function(x):
    return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Порожня функція

Якщо функція не має вмісту, використовуйте оператор `pass`:

```
def myfunction():
    pass
```

Позиційні аргументи

Щоб дозволити лише позиційні аргументи, додайте `, /` після параметрів:

```
def my_function(x, /):
    print(x)

my_function(3)
```

Іменовані аргументи

Щоб дозволити лише іменовані аргументи, додайте `*,` перед параметрами:

```
def my_function(*, x):
    print(x)

my_function(x=3)
```

Комбінування позиційних та іменованих аргументів

Ви можете комбінувати обидва типи аргументів:

```
def my_function(a, b, /, *, c, d):
    print(a + b + c + d)

my_function(5, 6, c=7, d=8)
```

Рекурсія

Функція може викликати саму себе. Це називається рекурсією:

```
def tri_recursion(k):
    if k > 0:
        result = k + tri_recursion(k - 1)
        print(result)
    else:
        result = 0
    return result

print("Результати рекурсії:")
tri_recursion(6)
```

Лямбда-функції

Лямбда-функція — це невелика анонімна функція, яка може мати будь-яку кількість аргументів, але лише один вираз:

```
x = lambda a: a + 10
print(x(5))
```

Лямбда-функції часто використовуються як анонімні функції всередині інших функцій:

```
def myfunc(n):
    return lambda a: a * n
```

```
mydoubler = myfunc(2)
print(mydoubler(11))
```

Використовуйте лямбда-функції, коли потрібна короткочасна анонімна функція.

Масиви в Python

> **Примітка:** Python не має вбудованої підтримки масивів, але можна використовувати списки (Lists) як їх заміну.

Масиви

> **Примітка:** Ця сторінка показує, як використовувати **списки** як **масиви**. Однак, для роботи з масивами в Python можна імпортувати бібліотеку, наприклад, `NumPy`.

Масиви використовуються для зберігання кількох значень в одній змінній:

Приклад

Створіть масив, що містить назви автомобілів:

```
cars = ["Ford", "Volvo", "BMW"]
```

Що таке масив?

Масив — це спеціальна змінна, яка може зберігати більше одного значення одночасно.

Наприклад, якщо у вас є список автомобілів, зберігання кожного автомобіля в окремій змінній виглядатиме так:

```
car1 = "Ford"
car2 = "Volvo"
car3 = "BMW"
```

Але що робити, якщо вам потрібно перебрати всі автомобілі або знайти конкретний? А якщо у вас не 3 автомобілі, а 300?

Рішенням є **масив**!

Масив може зберігати багато значень під одним ім'ям, і ви можете отримати доступ до значень за допомогою індексного номера.

Доступ до елементів масиву

Щоб звернутися до елемента масиву, використовуйте його індексний номер.

Приклад

Отримайте значення першого елемента масиву:

```
x = cars[0]
```

Приклад

Змініть значення першого елемента масиву:

```
cars[0] = "Toyota"
```

Довжина масиву

Використовуйте метод `len()`, щоб отримати кількість елементів у масиві.

Приклад

Отримайте кількість елементів у масиві `cars`:

```
x = len(cars)
```

> **Примітка:** Довжина масиву завжди на одиницю більша за найбільший індекс масиву.

Перебір елементів масиву

Використовуйте цикл `for` для перебору всіх елементів масиву.

Приклад

Виведіть кожен елемент масиву `cars`:

```
for x in cars:
    print(x)
```

Додавання елементів до масиву

Використовуйте метод `append()`, щоб додати елемент до масиву.

Приклад

Додайте ще один елемент до масиву `cars`:

```
cars.append("Honda")
```

Видалення елементів з масиву

Використовуйте метод `pop()`, щоб видалити елемент за індексом.

Приклад

Видаліть другий елемент масиву:

```
cars.pop(1)
```

Також можна використовувати метод `remove()`, щоб видалити елемент за значенням.

Приклад

Видаліть елемент зі значенням `"Volvo"`:

```
cars.remove("Volvo")
```

> **Примітка:** Метод `remove()` видаляє лише перше входження вказаного значення.

Методи масивів

Python має набір вбудованих методів, які можна використовувати зі списками/масивами:

Метод	Опис
`append()`	Додає елемент в кінець списку
`clear()`	Видаляє всі елементи зі списку
`copy()`	Повертає копію списку
`count()`	Повертає кількість елементів із вказаним значенням
`extend()`	Додає елементи зі списку (або іншого ітератора) в кінець поточного списку
`index()`	Повертає індекс першого елемента із вказаним значенням
`insert()`	Додає елемент у вказану позицію

Метод	Опис
`pop()`	Видаляє елемент за вказаним індексом
`remove()`	Видаляє перший елемент із вказаним значенням
`reverse()`	Змінює порядок елементів на зворотний
`sort()`	Сортує список

> **Примітка:** Python не має вбудованої підтримки масивів, але можна використовувати списки (Lists) як їх заміну.