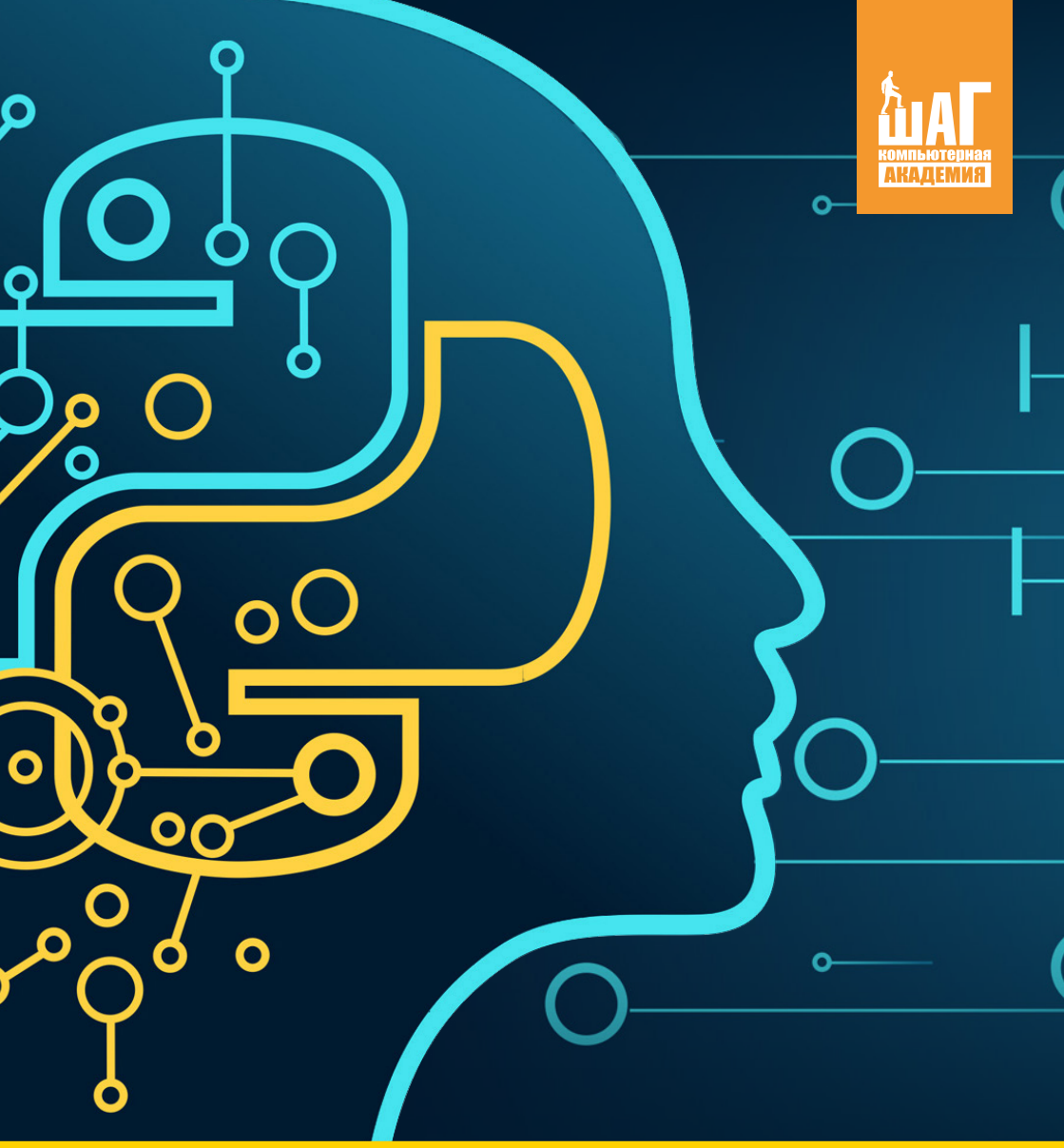

ШАГ
компьютерная
АКАДЕМИЯ

Основи програмування мовою Python

Урок 2

Оператори розгалужень, цикли, виключення

Зміст

1. Вступ

У попередньому модулі ми розглядали приклади, в яких інструкції виконувалися послідовно. Однак, іноді потрібно пропустити деякі інструкції або обрати інструкцію, яка буде виконана. У програмуванні, для надання варіативності використання та багатофункціональності, застосовуються умовні конструкції, конструкції для обробки винятків та цикли.

У цьому модулі будуть детально розглянуті:

- особливості роботи з операторами розгалуження і їх комбінування;
- особливості обробки винятків і їх типізація;
- цикли, їх типи і особливості використання.

2. Умовні інструкції та їх синтаксис

Оператори розгалужень (або умовні інструкції) дозволяють будувати прості конструкції, які перенаправляють виконання програми подібно до ліфта, що визначає загальну вагу пасажирів і на підставі цієї інформації починає переміщатися або повідомляти про перевантаження. Також, прикладом такої конструкції слугує сканер відбитків пальців, який дає або відмовляє в доступі, залежно від їх збігу з відбитками зареєстрованих користувачів.

У загальному випадку, для побудови умовної конструкції потрібно:

- вказати оператор розгалуження;
- розділити оператор та умову пробілом;
- вказати умову;
- поставити двокрапку в кінці умови;
- вказати набір інструкцій;
- виокремити набір інструкцій відступом.

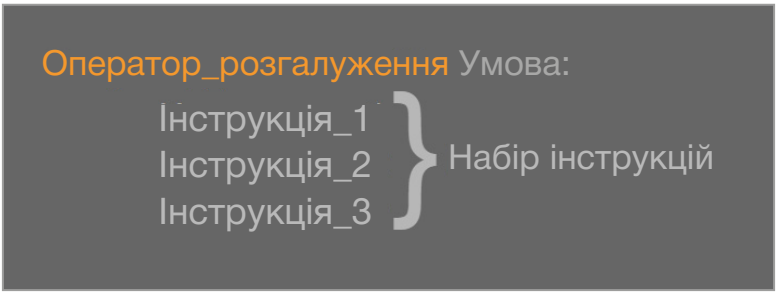

Рисунок 1

У програмуванні умови також називають логічними виразами. Але ми спочатку з'ясуємо, що являють собою логічні вирази та набори інструкцій.

2.1. Логічні вирази та оператори

Поряд з арифметичними та іншими операціями, у програмуванні використовуються операції порівняння, які дозволяють створювати логічні вирази на підставі порівнянь, і логічні операції, які дозволяють комбінувати декілька логічних виразів в один.

Логічні вирази — будь-які конструкції, результатом виконання яких є True або False.

Людська довіра демонструє роботу логічних виразів. Щоразу, коли ми чуємо якесь висловлювання, ми або віримо в це (тобто вважаємо за правду), або відносимося до цього з недовірою (тобто вважаємо хибним). Наприклад, якщо ваш друг покликав вас на прогулянку і сказав, що надворі гарна погода, може статися дві ситуації:

- ви визирнете у вікно і переконаєтеся, що погода дійсно хороша (тобто ствердження друга правдиве), і ви підете на прогулянку;
- ви визирнете у вікно і побачите, що погода погана (тобто ствердження друга хибне), і ви залишитеся вдома.

Звичайно, насправді ми можемо сумніватися, однак у програмуванні оцінка виразів завжди зводиться до одного з цих двох варіантів. Для уявлення істини або неправди використовуються ключові слова True та False.

2.1.1. Операції порівняння

Операції порівняння використовуються для створення умов спираючись на порівняння елементів. Елементи, до яких застосовується операція, називають операндами. Python підтримує наступні операції порівняння:

- `==` — «Дорівнює»: істинно (True), якщо обидва операнди рівні, інакше — хибне False;
- `!=` — «Не дорівнює»: істинно (True), якщо обидва операнди не рівні, інакше — хибне False;
- `>` — «Більше ніж»: істинно (True), якщо перший операнд більший за другий, інакше — хибне False;
- `<` — «Менше ніж»: істинно (True), якщо перший операнд менший за другий, інакше — хибне False;
- `>=` — «Більше або дорівнює»: істинно (True), якщо перший операнд більше або дорівнює другому, інакше — хибне False;
- `<=` — «Менше або дорівнює»: істинно (True), якщо перший операнд менше або дорівнює другому, інакше — хибне False.

```
print("1 == 1:", 1 == 1)      1 == 1: True
print("1 == 2:", 1 == 2)      1 == 2: False
print("1 != 1:", 1 != 1)      1 != 1: False
print("1 != 2:", 1 != 2)      1 != 2: True
print("1 > 1:", 1 > 1)        1 > 1: False
print("1 > 2:", 1 > 2)        1 > 2: False
print("2 > 1:", 2 > 1)        2 > 1: True
print("1 < 1:", 1 < 1)        1 < 1: False
print("1 < 2:", 1 < 2)        1 < 2: True
print("2 < 1:", 2 < 1)        2 < 1: False
```

```
print("1 >= 1:", 1 >= 1)      1 >= 1: True
print("1 >= 2:", 1 >= 2)      1 >= 2: False
print("2 >= 1:", 2 >= 1)      2 >= 1: True
print("1 <= 1:", 1 <= 1)      1 <= 1: True
print("1 <= 2:", 1 <= 2)      1 <= 2: True
print("2 <= 1:", 2 <= 1)      2 <= 1: False
```

На сірому фоні представлений код, який демонструє результати операцій порівняння з цілими числами. На чорному фоні відображається результат нашого коду в консолі.

2.1.2. Використання значень у якості умов

В Python існують правила, за якими можна привести будь-яке значення будь-якого типу до логічного. Так, якщо замість умови написати рядок або число, програма замінить його на True або False.

Значення, які будуть замінені на False:

- структури, які не містять елементів:
  - рядки без символів;
  - порожні списки, словники, масиви і т. д.
- нулі будь-яких чисельних типів:
  - 0;
  - 0.0.
- константа None.

Значення, які будуть замінені на True:

- структури, які містять елементи:
  - рядки будь-якої довжини, відмінної від нуля;
  - списки, словники, масиви і т. д. з елементами;

- не нульові значення будь-яких чисельних типів:
  - 1, 2, 3…;
  - 0.1, 1.2, 2.3… .

За допомогою функції bool() можна перевірити результат будь-якого такого перетворення:

```
print(bool(""))                          False
print(bool(0.0))                         False
print(bool(None))                        False
print(bool("IT Step Academy"))           True
print(bool(1))                           True
```

На сірому фоні представлений код, який демонструє результати операцій порівняння з цілими числами. На чорному фоні відображається результат нашого коду в консолі.

2.1.3. Логічні операції

Логічні операції дозволяють комбінувати декілька логічних виразів в один. В Python є такі логічні оператори:

- and

«Логічне множення»: повертає True, якщо обидва вирази дорівнюють True. Наприклад, роботодавець, який зацікавлений у хорошому співробітнику, візьме людину на роботу, якщо вона компетентна І відповідальна:

```
competent = True
responsible = True
print(competent and responsible)
```

Результат роботи нашого коду в консолі:

```
True
```

Якщо ж людина компетентна, але не відповідальна:

```
competent = True
responsible = False
print(competent and responsible)
```

Результат роботи нашого коду в консолі:

```
False
```

- or

«Логічне додавання»: повертає True, якщо хоча б один із виразів дорівнює True. Наприклад, роботодавець, який зацікавлений найняти співробітника у найкоротший термін, візьме людину на роботу, якщо вона компетентна АБО відповідальна:

```
competent = True
responsible = False
print(competent or responsible)
```

Результат роботи нашого коду в консолі:

```
True
```

Якщо ж людина не компетентна і не відповідальна:

```
competent = False
responsible = False
print(competent or responsible)
```

Результат роботи нашого коду в консолі:

```
False
```

- not

«Логічне заперечення»: повертає значення, зворотне операнду. Наприклад, роботодавець НЕ візьме людину, яка раніше була звільнена:

```
previously_fired = True
print(not previously_fired)
```

Результат роботи нашого коду в консолі:

```
False
```

В іншому випадку:

```
previously_fired = False
print(not previously_fired)
```

Результат роботи нашого коду в консолі:

```
True
```

Якщо один із операндів оператора and повертає False, то інший операнд не оцінюється, оскільки результатом виконання оператора у будь-якому випадку буде False. Аналогічно, якщо один із операндів оператора or повертає True, то другий операнд не оцінюється, тому що результат виконання оператора в будь-якому випадку буде True.

Розглянемо роботу логічних операцій на простому прикладі: повертаючись додому, особа захотіла скористуватися

метро. Вона пам'ятає, що метро працює з 6:00 до 24:00 (або 0). Щоб проїхати, їй необхідно пред'явити квиток або сплатити поїздку, крім того, її не пустять з великим багажем. На годинник особа ще не дивилася, квиток забула вдома, грошей на проїзд не вистачає і вона без багажу:

```
time = int(input("Enter the current time in hours:"))%24
ticket = False
money = True
luggage = False
print(money or ticket and not luggage and time > 6)
```

На перший погляд, наша умова вірна. Проте, вона істинна незалежно від значень time і luggage.

Класичний приклад з математики свідчить, тому що спочатку потрібно помножити, а потім додати. Такий спеціальний порядок обчислень визначено пріоритетом операцій. Таке є і в логічних операціях: спочатку виконується not, потім and і or. Саме тому or виконується наостанок і поверне True. Як і з математичними операціями, для перевизначення порядку виконання можна використовувати дужки:

```
print((money or ticket) and not luggage and time > 6)
```

На практиці, найпростішим прикладом застосування логічних виразів є if: Якщо умова істинна — виконується набір інструкцій:

```
car_speed = 100
if car_speed > 50:
   print("Car is faster than 50 km/h")
```

У цьому випадку, результатом виконання коду буде виведення на консоль рядка «*You have at least 3 flowers*», оскільки вираз 2 < 3 істинний.

```
passenger_weight = 400
if passenger_weight < 300:
    print("The elevator can go")
```

У цьому прикладі ліфт не зможе поїхати і рядок «*The elevator can go*» не буде виведений на консоль, оскільки вираз 400 < 300 помилковий.

Розглянемо приклад застосування логічних операцій з if:

```
car_speed = 100
if car_speed > 50 and car_speed < 150:
    print("Car speed is between 50 km/h and 150 km/h")
```

У цьому прикладі рядок «*Car speed is between 50 km/h and 150 km/h*» буде виведений у разі, якщо змінна «car_speed» перебуватиме у проміжку від 50 до 150 (не включно). Python дозволяє записати таку умову подібно до подвійної нерівності в математиці:

```
if 50 < car_speed < 150:
```

Відомо, що високосним роком вважається той рік, який кратний 4, не кратний 100 або кратний 400. Запишемо умовний вираз, який визначає, чи є рік високосним:

```
year = 2000
if year % 4 == 0 and year % 100 != 0 or year % 400 == 0:
    print("Year", year, "is leap")
```

2.2. Поняття «виконання блоку»

До цього ми розглядали загальний випадок використання оператора розгалуження, в якому згадувався «набір інструкцій». Однак, як визначити, що інструкція належить до цього набору або, іншими словами, до «блоку виконання»?

2.2.1. Визначення блоку виконання

В Python, на відміну від багатьох інших мов програмування, інструкції, які належать до одного блоку виконання, достатньо виділити однаковим відступом. Кінцем блоку виконання вважається остання інструкція, яка буде виділена відповідним відступом.

В іграх подібним чином визначаються комбінації або «комбо»: успішні дії, що йдуть одна за одною, утворюють «комбо», яке переривається, як тільки дія буде виконана невдало. Розглянемо приклад блоку виконання:

```
if 1 > 4:                                    # line 1
    print("This is the start
          of an execution block")            # line 2

    print("This is part
          of the execution block")           # line 3

    print("This is still part
          of the execution block")           # line 4

    print("This is the end
          of an execution block")            # line 5

    print("It is not part
          of the execution block")           # line 6
```

У цьому випадку результатом виконання коду буде виведення на консоль рядка «*It is not part of the execution block*». Оскільки вираз 1 > 4 хибний та інструкції, що належать до блоку виконання конструкції, не будуть виконані. Зокрема, до нього належать рядки від 2 до 5.

Відступи мають складатися з пробілів і, відповідно до інструкції з оформлення коду, їх кількість бажано робити кратною 4 (тобто 4 пробіли, 8, 16 ...). Хоча, якщо блок буде виділений відступом у 2 пробіли, програма також працюватиме. Крім того, в останніх версіях мови для відступу не можна використовувати табуляцію, проте інтегровані середовища розробки (у скороченому варіанті «IDE») автоматично розміщують 4 пробіли при натисканні кнопки «*Tab*» на клавіатурі.

2.2.2. Застосування

Визначення блоків виконання необхідне при використанні умовних конструкцій. Вони також використовуються в:

- циклах;

```
while condition:
    print("Cycles")
```

- обробці винятків;

```
try:
    print("Some code")
except:
    print("Error processing")
finally:
    print("Handling Exceptions")
```

- функціях;

```
def function():
    print("Functions")
```

- класах.

```
class Class:
    def __init__(self):
        print("Classes")
```

З цими конструкціями ви ознайомитеся пізніше.

2.3. Оператори розгалуження if …else

Тепер, коли ви навчилися виокремлювати інструкції в блоки, складати і комбінувати вирази, можна розпочати розгляд умовних конструкцій. Раніше був наведений найпростіший приклад використання умовної конструкції if:

```
car_speed = 100
if car_speed > 50:
    print("Car is faster than 50 km/h")
```

Спробуймо розширити цей приклад спираючись на отримані знання. Введемо дві змінні, що позначають швидкість (в км/год) машини і мотоцикла відповідно:

```
car_speed = 150
motorcycle_speed = 100
```

У такому разі наш приклад можна змінити за допомогою змінних:

```
if car_speed > motorcycle_speed:
    print("Car is faster than motorcycle")
```

Результат роботи нашого коду в консолі:

```
Car is faster than motorcycle
```

Якщо змінити швидкість так, що машина виявиться швидшою, наша умова виявиться помилковою і користувач не отримає жодної інформації. У такому разі зручно використовувати else — блок, який виконується лише якщо остання умова виявилася хибною. Додаємо його до нашого прикладу:

```
car_speed = 100
motorcycle_speed = 150
if car_speed > motorcycle_speed:
   print("Car is faster than motorcycle")
else:
   print("Motorcycle is faster than car")
```

Результат роботи нашого коду в консолі:

```
Motorcycle is faster than car
```

Швидкості можуть зрівнятися і, хоча блок else виконається, представлена користувачеві інформація буде неправильною. Виправити це можна, додавши ще одну умову:

```
car_speed = 100
motorcycle_speed = 100
if car_speed > motorcycle_speed:
    print("Car is faster than motorcycle")
```

```
if motorcycle_speed > car_speed:
    print("Motorcycle is faster than car")
else:
    print("Car and motorcycle are equally fast")
```

Результат роботи нашого коду в консолі:

```
Car and motorcycle are equally fast
```

Тут умовні конструкції виконуються по черзі: спочатку перевіряється умова motorcycle_speed < car_speed, потім motorcycle_speed > car_speed. Зверніть увагу, що друга умова буде перевірена незалежно від істини першої. Корисність такої поведінки ситуативна: якщо ці конструкції самостійні і передбачають як незалежне, так і спільне виконання — це корисно, проте якщо вони пов'язані і передбачається виконання лише однієї конструкції — це може призвести до помилок. Цей приклад добре підходить для порівняння постійних швидкостей. Якщо за час порівняння швидкість мотоцикла зміниться, ми отримаємо два суперечливі результати:

```
car_speed = 130
motorcycle_speed = 100
if car_speed > motorcycle_speed:
   print("Car is faster than motorcycle")
   motorcycle_speed += 50
if motorcycle_speed > car_speed:
   print("Motorcycle is faster than car")
   motorcycle_speed += 50
else:
   print("Car and motorcycle are equally fast")
   motorcycle_speed += 50
```

Результат роботи нашого коду в консолі:

```
Car is faster than motorcycle
Motorcycle is faster than car
```

Так, при першому порівнянні умова motorcycle_speed < car_speed буде істинною, швидкість мотоцикла зміниться, і при другому порівнянні motorcycle_speed > car_speed також виявиться істинною. Уникнути такої ситуації можна, об'єднавши дві умови в одну конструкцію. Для цього замінимо другий if на elif — комбінацію блоків else і if, набір інструкцій якого виконується, якщо умови попередніх блоків помилкові, а власна умова блоку є істинною:

```
if car_speed > motorcycle_speed:
    print("Car is faster than motorcycle")
    motorcycle_speed += 50
elif motorcycle_speed > car_speed:
    print("Motorcycle is faster than car")
    motorcycle_speed += 50
else:
    print("Car and motorcycle are equally fast")
    motorcycle_speed += 50
```

Результат роботи нашого коду в консолі:

```
Car is faster than motorcycle
```

2.4. Вкладені конструкції

Блоки виконання if, elif та else можуть містити інші умовні конструкції, які називають «вкладеними». Блок виконання вкладених конструкцій також має бути

відділений відступом, що утворює свого роду новий «крок»:

```
flowers_amount = 3
if flowers_amount > 2:
    print("You have at least 3 flowers")
    if flowers_amount < 5:
        print(""You have less than 5 flowers")
```

За допомогою вкладених конструкцій можна продемонструвати принцип роботи блоку elif. Припустимо, що користувач проходить тест і має вибрати один із варіантів відповіді, ввівши його номер:

```
number = int(input("Enter the answer number: "))

if number == 1:
   print("You've chosen answer A")
else:
   if number == 2:
       print("You've chosen answer B")
   else:
       if number == 3:
           print("You've chosen answer C")
       else:
           if number == 4:
               print("You've chosen answer D")
           else:
               print("There is no such answer.")
```

Результат роботи нашого коду в консолі:

```
Enter the answer number: 4
You've chosen answer D
```

Такий код, побудований з використанням вкладеності, можна спростити, використовуючи elif. При цьому, результат виконання коду не зміниться:

```
number = int(input("Enter the answer number: "))

if number == 1:
   print("You've chosen answer A")
elif number == 2:
   print("You've chosen answer B")
elif number == 3:
   print("You've chosen answer C")
elif number == 4:
   print("You've chosen answer D")
else:
   print("There is no such answer.")
```

Розглянемо практичне застосування вкладених конструкцій на прикладі скарбнички:

```
account = int(input("Enter how much you put: "))
account = abs(account)

if account > 0:
    withdrawal = int(input("Enter how much you take:"))
    withdrawal = abs(withdrawal)

    if withdrawal < account:
        account -= withdrawal
        print("Here are your", withdrawal, ".")
        print("There are", account, "left.")
    else:
        print("There are only", account, ".")
else:
    print("There are no money in piggy bank")
```

У цьому прикладі для введення користувачем суми, яку він поклав у скарбничку, використовують рядки:

```
account = int(input("Enter how much you put: "))
account = abs(account)
```

Причому, за отримання від користувача рядка та приведення його до цілого числа відповідає перший рядок, а другий рядок використовує функцію abs(), результатом якої є модуль переданого їй числа. Подібна конструкція використовується для отримання суми, яку користувач хоче забрати зі скарбнички.

З прикладу видно, що забрати гроші (незалежно від того, чи достатньо їх чи ні) можна, тільки якщо вони є в скарбничці. Цей приклад можна спростити, використовуючи розглянуті раніше правила приведення будь-яких типів до логічного. Оскільки account — позитивне число (завдяки використанню функції abs()), а нульові значення чисел зводяться до False, ми можемо замінити account > 0 на account не змінивши при цьому поведінку програми.

3. Винятки

Винятки — один із двох основних типів помилок у програмуванні. На відміну від синтаксичних помилок, які виникають під час написання, винятки можуть виникнути під час виконання програми. Прикладом такої відмінності може бути автомобіль: він може бути несправний, що зробить подорож на ньому неможливою (подібно до синтаксичних помилок), крім того, водій може не впоратися з керуванням, що призведе до ДТП вже під час поїздки (подібно до винятків). У програмуванні винятки призводять не до ДТП, а до повного припинення або неправильного виконання програм. Для уникнення припинення роботи або отримання додаткової інформації про помилку використовують конструкції обробки винятків.

3.1. Типи винятків

Кожний виняток має власний тип, який визначається тим, яка помилка його викликала, і дає можливість по-різному реагувати на різні види помилок.

Розглянемо деякі типи винятків, які можуть виникнути під час цього курсу:

- BaseException — базовий тип, з якого походять всі інші, зокрема системні:
  - Exception — базовий тип для «стандартних» і користувальницьких винятків:
    - ArithmeticError — арифметична помилка:

    - OverflowError — виникає, коли результат арифметичної операції дуже великий для уявлення;

На сьогоднішній день практично неможливо отримати OverflowError використовуючи стандартну бібліотеку Python, оскільки цілі числа мають динамічну довжину і викличуть MemoryError, а числа з плаваючою комою при перетині граничних значень замінюються на 0.0 або inf. Проте, ця помилка може бути отримана при роботі зі сторонніми бібліотеками або модулями, написаними мовами програмування з жорсткою типізацією (числові типи мають чіткі межі, вихід за які буде помилкою).

- ZeroDivisionError — ділення на нуль.
- ImportError — імпортувати модуль або його атрибут не вдалося;
- LookupError — некоректний індекс або ключ:
  - IndexError — індекс не входить до діапазону елементів;
  - KeyError — неіснуючий ключ (наприклад, у словнику).
- NameError — не знайдено змінної із зазначеним ім'ям;
- RuntimeError — виникає, коли виняток не підпадає під жодну з інших категорій;
- SyntaxError — синтаксична помилка:
  - IndentationError — неправильні відступи:
    - TabError — змішування у відступах табуляції і пробілів.
- TypeError — операцію застосовано до об'єкта невідповідного типу;
- ValueError — функція набуває аргументу правильного типу, але некоректного значення.

Список передбачених мовою програмування («вбудованих») типів винятків набагато ширший і міститься в документації до мови програмування. Крім того, користувачі можуть визначати нові типи винятків у своїх програмах і бібліотеках.

3.2. Перехоплення винятків

Для обробки винятків в Python використовують конструкцію «try …except». Загалом, для побудови цієї конструкції необхідно:

- відкрити блок «try», ввівши відповідну інструкцію і двокрапку;
- вказати набір інструкцій, в результаті яких може виникнути виняток;
- відступом виділити набір інструкцій;
- відкрити блок «except», ввівши відповідну інструкцію і двокрапку;
- вказати набір інструкцій, які потрібно виконати у разі виключення;
- виокремити набір інструкцій відступом.

Застосування цієї конструкції розглянемо на такому прикладі: на склад доставили 10 ящиків, на складі також є бочки. Програма обліку предметів, що зберігаються на складі, просить користувача вказати, скільки і які саме предмети були доставлені:

```
amount = int(input("Enter the amount of received
                    items: "))
items_type = input("Specify the type of received
                    items: ")
```

Оскільки прохання вказати тип отриманих елементів, відображається лише після введення їх числа, користувачі, які не знайомі з програмою, можуть припуститися помилки:

```
Enter the amount of received items: 10 boxes
```

Частину введеного рядка «boxes» не можна привести до цілого числа, внаслідок чого буде викликано виняток «ValueError».

```
Enter the amount of received items: 10 boxes
Traceback (most recent call last):
  File "C:/Users/[redacted]/PycharmProjects/Module_2/main.py", line 331, in <module>
    amount = int(input("Enter the amount of received items: "))
ValueError: invalid literal for int() with base 10: '10 boxes'

Process finished with exit code 1
```

Рисунок 2

Для обробки цього виключення застосуємо конструкцію «try …except»:

```
try:
    amount = int(input("Enter the amount of received
                        items: "))
    items_type = input("Specify the type of received
                        items: ")
except:
    print("Amount should be an integer")
```

Результат роботи нашого коду в консолі:

```
Enter the amount of received items: 10 boxes
Amount should be an integer
```

Так, якщо користувач введе рядок, який не можна привести до цілого числа, виконання блоку «try» буде перервано (усі наступні інструкції блоку виконання будуть пропущені), і виконання перейде до блоку «except». Припустимо, програма обліку поділяє отримані предмети на декілька рівних частин, щоб їх можна було розташувати по різним рівням складу. додаємо до блоку «try» наступні інструкції:

```
parts_number = int(input("Enter the number of parts:"))
parts = amount / parts_number
```

Якщо користувач як «parts_number» вкаже «0» — буде помилка, яка відповідає типу виключення «ZeroDivisionError» (ділення на нуль неможливе). Однак повідомлення про помилку, показане користувачеві, залишиться незмінним. Для того, щоб при різних помилках виконувались різні інструкції, необхідно створити декілька блоків «except» і вказати типи винятків, за яких вони виконуватимуться:

```
try:
    amount = int(input("Enter the amount of received
                        items: "))
    items_type = input("Specify the type of received
                        items: ")
    parts_number = int(input("Enter the number
                              of parts: "))
    parts_amount = amount / parts_number
    print("Supply of", amount, items_type, "saved")
    print("Each of", parts_number, "parts consists of",
          parts_amount, items_type)
```

```
except ValueError:
    print("Amount should be an integer")

except ZeroDivisionError:
    print("You cannot divide the delivery into 0 parts")
```

Результат роботи нашого коду в консолі:

```
Enter the amount of received items: 10
Specify the type of received items: boxes
Enter the number of parts: 0
You cannot divide the delivery into 0 parts
```

Крім того, блок «finally» передбачений конструкцією «try ... except», інструкції якого будуть виконані незалежно від того, чи виникне виняток:

```
try:
    amount = int(input("Enter the amount of received
                        items: "))
    items_type = input("Specify the type of received
                        items: ")
    parts_number = int(input("Enter the number of
                              parts: "))
    parts_amount = amount / parts_number
    print("Supply of", amount, items_type, "saved")
    print("Each of", parts_number, "parts consists of",
          parts_amount, items_type)

except ValueError:
    print("Amount should be an integer")
except ZeroDivisionError:
    print("You cannot divide the delivery into 0 parts")
finally:
    print("The program has finished")
```

Блок «finally» використовується у тому випадку, коли нам необхідно гарантувати закінчення роботи програми. Наприклад, ми можемо підключитися до віддаленого сервера мережі, працювати з файлом або графічним інтерфейсом користувача (GUI). У всіх цих випадках ми повинні звільнити ресурси, які використовуються програмою до того, як програма завершиться незалежно від того, чи успішно вона спрацювала. Такі дії (звільнення ресурсів) виконуються у блоці «finally». Розглянемо загальний випадок застосування на прикладі роботи з файлом:

```
try:
  f = open("test.txt",'w')
  # perform file operations
finally:
  f.close()
```

3.3. Особливості роботи з try ...except

3.3.1. Виклик винятків

В Python ми можемо не тільки перехоплювати винятки, але й викликати їх напряму. Для цього необхідно використати ключове слово «raise». За допомогою «raise» ми можемо вказати, до якого типу належатиме викликаний виняток:

```
try:
    apples = int(input("Enter the amount of apples
                       you have: "))
    if apples < 0:
        raise Exception
    print("You have", apples, "apples")
```

```
except Exception:
    print("You can't have -10 apples")
```

Результат роботи нашого коду в консолі:

```
Enter the amount of apples you have: -10
You can't have -10 apples
```

Виклик виключень може знадобитися у випадках, коли логіка програми потребує додаткових умов. Наприклад, у наведеному вище прикладі для програми не важливо, буде введено 10 або -10, оскільки це цілі числа. Однак, ми з вами знаємо, що для позначення кількості предметів використовуються натуральні числа (1, 2, 3…) і нуль. Отже, яблук не може бути -10.

3.3.2. Порядок розміщення блоків exept

Однією з головних особливостей роботи з «try …except» є правильне розташування блоків «except». У попередньому прикладі з «ValueError» та «ZeroDivisionError» розташування не відігравало ролі, оскільки жоден із них не є підтипом іншого. З таблиці типів винятків, наведеної у відповідному розділі, видно, що «ZeroDivisionError» є підтипом «ArithmeticError», а «ValueError» — підтипом «Exception». Повну структуру ієрархії вбудованих типів винятків можна переглянути в документації до мови програмування. Розглянемо вплив розташування блоків на роботу програми з прикладу пари «ValueError» та «Exception»:

```
try:
    raise Exception
```

```
except Exception:
    print("Hmm... Something went wrong")
except ValueError:
    print("Improper value was obtained")
```

Результат роботи нашого коду в консолі:

```
Hmm... Something went wrong
```

У цьому прикладі, всередині блоку try виникає виняток Exception, який обробляється відповідним блоком except. Змінимо тип викликаного виключення:

```
try:
    raise ValueError
except Exception:
    print("Hmm... Something went wrong")
except ValueError:
    print("Improper value was obtained")
```

Результат роботи нашого коду в консолі:

```
Hmm... Something went wrong
```

«ValueError» є підтипом «Exception». Виниклий виняток буде оброблений першим блоком, а другий, призначений для обробки виключень такого типу, виконаний не буде. Для того, щоб виправити це, необхідно змінити послідовність блоків except. Загальні типи винятків мають розташовуватися нижче приватних:

```
try:
    raise ValueError
```

```
except ValueError:
    print("Improper value was obtained")
except Exception:
    print("Hmm... Something went wrong")
```

Результат роботи нашого коду в консолі:

```
Improper value was obtained
```

3.3.3. Комбінування загального і конкретних блоків exept

Ми також можемо комбінувати використання блоків «exept» для конкретних типів винятків та загальний блок «exept». В такому випадку загальний блок має бути розміщений останнім. Він виконається, якщо тип отриманого виключення не збігається з жодним типом блоків «exept»:

```
try:
    f = open("Some_file.txt")
except ZeroDivisionError:
    print("You cannot divide the delivery into 0 parts")
except:
    print("Hmm... Something went wrong")
```

Результат роботи нашого коду в консолі:

```
Hmm... Something went wrong
```

У цьому прикладі ми намагаємося відкрити файл «Some_file.txt», який не був створений раніше, що призводить до виключення типу «FileNotFoundError». Однак, для цього типу не передбачено окремого блоку «exept», тому виключення обробляється загальним блоком.

3.3.4. Блок exept для кількох типів винятків

Python також дозволяє один блок «exept» для різних типів винятків. Для цього їх необхідно розмістити через кому всередині круглих дужок. Щоб розглянути таку особливість, трохи видозмінимо один із попередніх прикладів:

```
try:
    amount = int(input("Enter the amount
                        of received items: "))
    items_type = input("Specify the type of received
                        items: ")
    parts_number = int(input("Enter the number
                              of parts: "))
    parts_amount = amount / parts_number
    print("Supply of", amount, items_type, "saved")
    print("Each of", parts_number, "parts consists of",
          parts_amount, items_type)

except (ValueError, ZeroDivisionError):
    print("Improper value was obtained")
```

З огляду на те, що помилки є об'єктами відповідних типів (докладніше про визначення об'єкта розповімо у наступних модулях), їх можна присвоїти змінним за допомогою ключового слова «as»:

```
try:
    x = 1/0
except Exception as ex:
    print(ex)
```

У цьому прикладі об'єкт помилки присвоюється змінною «ex». Результатом виконання цього коду буде виведення додаткової інформації про помилку, що зберігається у

змінній. Використовуючи ключове слово «raise», можна вручну вказати, яке повідомлення зберігатиметься в об'єкті виключення:

```
raise ValueError("You made a mistake entering a value")
```

Крім того, використовуючи змінну, в якій зберігається об'єкт виключення, можна отримати точну інформацію про тип виключення:

```
try:
   raise ValueError
except BaseException as ex:
   print(type(ex).__name__)
```

Наведений код демонструє, що під час виконання інструкцій блоку «except», на консоль буде виведено рядок «ValueError», який представляє тип виниклого виключення.

Підсумуємо отримані знання на наступному прикладі:

```
try:
   apples = int(input("Enter the amount
                       of apples you have: "))
   if apples < 0:
       raise Exception("You can't have -10 apples")
   parts_number = int(input("Enter the number
                              of parts: "))
   parts_amount = apples / parts_number
   print("You have " + str(apples) + " apples \n")
   print("Each of " + str(parts_number) +
         " parts consists of " + str(parts_amount) +
         " apples")
except (ZeroDivisionError, ValueError):
   print("Improper value was obtained")
```

```
except Exception as ex:
   print(ex)
except:
   print("Hmm... Something went wrong")

finally:
    print("The program has finished")
```

4. Цикли

Щодня ми стикаємося з циклами — процесом виконання однотипних дій. Хорошим прикладом циклу є тренування, коли спортсмену необхідно виконати певний рух декілька разів для успішного виконання вправи. Навіть підйом сходами так само є циклом, тому що для цього необхідно кілька разів піднятися на наступну сходинку. Циклічність має властивість закінчуватися. Відтак, досягнувши потрібного поверху, людина не підніматиметься на наступну сходинку, а спортсмен закінчить вправу, коли виконає необхідну кількість повторень.

4.1. Поняття ітерації

Ітерація — окреме повторення однотипної дії, під якою ми розуміємо блок виконання циклу («тіло циклу»). Повертаючись до прикладу зі сходами, ітерацією вважатиметься підйом на нову сходинку.

4.2. Цикл while

Насамперед розглянемо цикл «while», синтаксис якого ідентичний конструкції «if». Такий цикл повторює виконання тіла циклу доти, доки відповідна умова не виявиться хибною:

```
number = 1
while number < 3:
    print(number, "is greater than 0 and less than 3")
    number += 1
```

Результат роботи нашого коду в консолі:

```
1 is greater than 0 and less than 3
2 is greater than 0 and less than 3
```

Розглянемо роботу циклу покроково:

- Створення змінної number" зі значенням 1;
- Перевірка умови циклу «number < 3»: оскільки умова є істинною, виконується тіло циклу;
- Виконується тіло циклу: на консоль виводиться рядок, значення «number» збільшується на 1;
- Перевірка умови циклу «number < 3»: оскільки умова є істинною, виконується тіло циклу;
- Виконується тіло циклу: на консоль виводиться рядок, значення «number» збільшується на 1;
- Перевірка умови циклу «number < 3»: оскільки умова помилкова, виконання циклу і програми закінчено.

Інакше кажучи, тіло циклу, яке представлене рядками 3 та 4, повториться двічі, оскільки після другого повторення змінна «number» матиме значення 3 і умова «number < 3» виявиться хибною. Зверніть увагу, що перевірка істинності умови циклу виконується лише після виконання останньої інструкції тіла. Якщо поміняти місцями рядки 3 та 4, цикл також виконається двічі, але представлена інформація для користувача виявиться хибною, оскільки на момент виконання «print()» значення «number» вже не задовольнятиме умові:

```
number = 1
while number < 3:
    number += 1
    print(number, "is greater than 0 and less than 3")
```

Результат роботи нашого коду в консолі:

```
2 is greater than 0 and less than 3
3 is greater than 0 and less than 3
```

Спробуємо знайти суму всіх цілих чисел, розташованих між «x1» та «x2». Для цього використовуємо цикл «while» наступним чином:

```
x1 = int(input("Enter x1: "))
x2 = int(input("Enter x2: "))
x = x1 + 1
sum = 0

while x < x2:
    sum += x
    x += 1
print("The sum of all integers between", x1, "and",
      x2, "is", sum)
```

Результат роботи нашого коду в консолі:

```
Enter x1: -13
Enter x2: 17
The sum of all integers between -13 and 17 is 58
```

У цьому прикладі ми створюємо змінну «x», якій присвоюємо перше ціле число після «x1» (“x1 + 1”), і «sum» для зберігання проміжного результату підсумовування. Цикл розпочнеться лише в тому випадку, якщо у проміжку від «x1» до «x2» є хоча б одне ціле число. В інакшому випадку, умова «x < x2» не буде виконаною, цикл буде пропущено, і сума так і залишиться 0. У кожній

ітерації циклу ми додаємо до проміжного результату («sum») «x» (одне з чисел у проміжку від «x1» до «x2»), після чого замінюємо його на наступне (“x + 1”). Коли до «sum» буде додано останнє значення проміжку від «x1» до «x2», його подальше збільшення (“x + 1”) призведе до того, що умова «x < x2» виявиться хибною і цикл буде завершено. Таким чином, у змінній «sum» будуть послідовно підсумовані усі числа у зазначеному користувачем проміжку.

4.3. Цикл for

Цикли типу «for», на відміну від «while», повторюються не залежно від виконання умови, а для кожного елемента у списку, множині, кортежі або іншій сукупності елементів. Інакше кажучи, для кожної ітерації циклу використовується один із елементів сукупності. Наприклад, щоб перевірити придатність продуктів харчування в магазині, людина має послідовно взяти кожен продукт зі свого кошика і переконатися, що термін придатності ще не минув. Так, кошик із продуктами виступає як сукупність елементів. Продукти є елементами, а перевірка терміну придатності одного продукту — ітерацією циклу.

Рядок може виступати як сукупність елементів, оскільки він є сукупністю символів. Спробуємо вивести усі елементи рядка окремо:

```
line = input("Enter some string: ")
for c in line:
    print(c)
```

Результат роботи нашого коду в консолі:

```
Enter some string: ItStep
I
t
S
t
e
p
```

Цикл може пройти рядками частково. І тому, необхідно розділити строку за допомогою оператора зрізу «[::]». «line[start, stop, step]» повертає усі елементи рядка «line» від «start» (включно) до «stop» (не включно). Крок визначає, як і наскільки збільшуватиметься індекс. Іншими словами, крок вказує на те, який символ рядка буде прийнято за наступний. Значення «start» за замовчуванням — 0, «step» — 1, «stop» за замовчуванням відповідає індексу останнього елемента рядка. Детальніше, цей оператор буде розглянуто у наступних модулях, тому зупинимося на тих його властивостях, які знадобляться нам для презентації робіт. Нижче наведено приклад коду, який виконує ітерацію за першими шістьма літерами рядка:

```
line = input("Enter some string: ")
for c in line[0 : 6 : 1]:
    print(c)
```

Результат роботи нашого коду в консолі:

```
Enter some string: ItStepAcademy
I
t
```

```
S
t
e
p
```

В «[0:6:1]» 0 означає, що зріз починається з початку рядка, 6 — індекс останнього елемента у зрізі не перевищує 6 (у нашому випадку, 6-й символ — перша «s»), а 1 означає, що символи у зріз будуть додаватися зліва направо без пробілу. 0 і 1 у прикладі вище — значення, які використовуються оператором за замовчуванням, тому надалі значення за замовчуванням будуть пропущені. Змінюючи значення кроку, можна вивести усі символи, які стоять на парних позиціях:

```
line = input("Enter some string: ")

for c in line[ : : 2]:
   print(c)
```

Результат роботи нашого коду в консолі:

```
Enter some string: ItStepAcademy
I
S
e
A
a
e
y
```

У прикладі «[: : 2]» означає, що у зрізі буде пропущений кожний другий символ початкового рядка. Крім

того, якщо використовувати негативне значення як крок, перебір відбуватиметься у зворотний бік:

```
line = input("Enter some string: ")
for c in line[ : : -1]:
    print(c)
```

Результат роботи нашого коду в консолі:

```
Enter some string: ItStep
p
e
t
S
t
I
```

«[: : -1]» означає, що рядок буде інвертовано: його кінець стане початком, а початок — кінцем.

4.4. Нескінченні цикли

Визначення циклу «while» говорить наступне: «такий цикл повторює виконання тіла циклу до тих пір, доки відповідна умова не виявиться хибною». Неважко уявити ситуацію, в якій умова так і не виявиться хибною. У такому разі цикл виконуватиметься нескінченно:

```
while True:
    print("Hello!")
```

Результат роботи нашого коду в консолі:

```
Hello!
Hello!
Hello!
Hello!
...
```

Найчастіше нескінченні цикли є результатом помилки написання програми (програма складена таким чином, що передбачена умова виходу із циклу ніколи не виконається), наприклад:

```
number = 0
while number != 15:
    number += 2
    print(number, "is still not 15")
```

Результат роботи нашого коду в консолі:

```
...
10 is still not 15
12 is still not 15
14 is still not 15
16 is still not 15
18 is still not 15
...
```

Оскільки «number» спочатку дорівнює 0 і в ході виконання програми залишається парним, він ніколи не стане дорівнювати 15 і не зробить умову помилковою.

Нескінченні цикли можуть застосовуватися для обробки серверами клієнтських запитів або, наприклад, щоб продемонструвати нескінченні величини:

```
import time
number = 1
while True:
    print(number, "is natural number")
    number += 1
    time.sleep(0.1)
```

Результат роботи нашого коду в консолі:

```
1 is natural number
2 is natural number
3 is natural number
4 is natural number
5 is natural number
...
```

У цьому прикладі інструкція «import time» необхідна для використання «time.sleep()», яка, у свою чергу, зупиняє виконання програми на 0.1 секунду, аби користувач встиг побачити виведені на екран цифри. Ви можете приблизно оцінити швидкість виконання інструкцій програмою, прибравши ці дві інструкції.

4.5. Вкладені конструкції

Подібно до умовних конструкцій, цикли також можуть містити в собі інші цикли та конструкції. Розглянемо приклад виведення таблиці множення:

```
for i in range(1, 10):
    for j in range(1, 10):
        print(i * j, end="\t")
    print("\n")
```

Результат роботи нашого коду в консолі:

```
1    2    3    4    5    6    7    8    9
2    4    6    8    10   12   14   16   18
3    6    9    12   15   18   21   24   27
4    8    12   16   20   24   28   32   36
5    10   15   20   25   30   35   40   45
6    12   18   24   30   36   42   48   54
7    14   21   28   35   42   49   56   63
8    16   24   32   40   48   56   64   72
9    18   27   36   45   54   63   72   81
```

Тут внутрішній цикл виконується 9 разів на кожній ітерації зовнішнього циклу, яких теж 9. Таблиця множення виходить вдало, оскільки у тілі внутрішнього циклу ми виводимо значення елементів обох циклів.

Розглянемо ще один приклад: людині необхідно піднятися на 5-й поверх. Наразі вона знаходиться на першому поверсі, між кожними двома поверхами по 20 сходинок. Пройшовши 70 сходинок поспіль, людина втомлюється і зупиняється, щоб відпочити:

```
floor = 1
energy = 70
print("I'm on the", floor, "floor")
while floor != 5:
    step = 0
    while step != 20:
        step += 1
        energy -= 1
        if energy == 0:
            print("I'm tired, I will rest a little")
```

```
            energy += 70
    floor += 1
    print("Now I'm on the", floor, "floor")
print("I've got to the right floor")
```

Результат роботи нашого коду в консолі:

```
I'm on the 1 floor
Now I'm on the 2 floor
Now I'm on the 3 floor
Now I'm on the 4 floor
I'm tired, I will rest a little
Now I'm on the 5 floor
I've got to the right floor
```

Наведений код демонструє, що кожна ітерація циклу «while floor != 5:» виконує вкладений цикл, а вкладений цикл виконує конструкцію «if».

4.6. Керуючі оператори continue, break, else

Для керування виконанням циклу використовуються оператори «continue», «break» та «else», які дозволяють пропустити ітерацію або виконати набір інструкцій після успішного завершення відповідного циклу. Розглянемо приклади їх використання.

4.6.1. break

Оператор «break» використовується для переривання виконання циклу.

```
x = 1
while x < 10:
```

```
    print(x)
    x += 1
    break
```

Результат роботи нашого коду в консолі:

```
1
```

У цьому прикладі на консоль виведено лише «1», оскільки у першій ітерації виконався оператор «break» і цикл був перерваний.

Якщо у прикладі зі сходами, людина, піднявшись на 3-й поверх, скористається ліфтом, код можна змінити так:

```
floor = 1
energy = 70
print("I'm on the", floor, "floor")

while floor != 5:
    step = 0
    if floor == 3:
        print("I will take an elevator")
        break
    while step != 20:
        step += 1
        energy -= 1
        if energy == 0:
            print("I'm tired, I will rest a little")
            energy += 70
    floor += 1
    print("Now i'm on the", floor, "floor")

print("I've got to the right floor")
```

Результат роботи нашого коду в консолі:

```
I’m on the 1 floor
Now i’m on the 2 floor
Now i’m on the 3 floor
I will take an elevator
I’ve got to the right floor
```

Тут, дійшовши до 3-го поверха, використовується оператор «break», який перериває цикл підйому сходами. Зверніть увагу: під час використання оператора «break» у вкладеному циклі, зовнішній цикл перервано не буде:

```
number = 0
while number < 5:
    print(number)
    while True:
        number += 1
        break
        number -= 1
```

Результат роботи нашого коду в консолі:

```
0
1
2
3
4
```

У наведеному прикладі також бачимо, що за допомогою «break» можна перервати нескінченний цикл. Ця особливість може бути корисною для створення циклів з пост-умовою. Ви вже могли помітити, що цикл «while» виконує перевірку умови перед виконанням тіла циклу,

тому можлива ситуація, коли умова буде хибною і тіло циклу не виконається у жодному разі. В інших мовах, для уникнення подібної ситуації можна використовувати цикл «do …while» («цикл із пост-умовою»), в якому спочатку виконується тіло циклу, а потім перевіряється умова. В Python такого циклу немає, але його нескладно створити самому. Припустимо, що ми хочемо зробити цикл із пост-умовою з іншого, раніше створеного циклу «while»:

```
number = 1
while number < 5:
    print(number)
    number += 1
```

Результат роботи нашого коду в консолі:

```
1
2
3
4
```

Для цього замінимо умову циклу на «while», що зробить його нескінченним, і помістимо оператор «break» в умовну конструкцію, яка використовує умову в кінці тіла циклу, протилежну первісному циклу:

```
number = 1
while True:
    print(number)
    number += 1
    if number >= 5:
        break
```

Результат роботи нашого коду в консолі:

```
1
2
3
4
```

Якби ми використовували початкову умову завершення циклу «number < 5», оператор «break» припиняв би виконання циклу всім значенням «number», які нам підходять (1, 2, 3, 4, тобто < 5), і робив би цикл нескінченним усім невідповідним значенням (>= 5). Тому, умову необхідно замінити на протилежну. Інакше кажучи, у початковому циклі умова вказувала на необхідність продовження повторень, а в циклі з пост-умовою нам потрібно, щоб вона вказувала на необхідність їх припинення. Хорошим прикладом для розуміння такої поведінки є сама мова. Одну фразу ми можемо передати двома способами:

- Я продовжуватиму радіти, доки мені не виповниться 100 років.
- Я припиню радіти, коли мені виповниться 100 років.

У нашому випадку очевидно, що протилежну умову можна отримати, використовуючи інший оператор порівняння («>=»), але для складніших умов розумніше змінювати умову, використовуючи логічний оператор «not»:

```
if not (number < 5):
```

У прикладі отриманого циклу ми бачимо, що його тіло виконається один раз, навіть якщо умова «number < 5» на початку помилкова:

```
number = 5

while True:
    print(number)
    number += 1
    if not (number < 5):
        break
```

Результат роботи нашого коду в консолі:

```
5
```

4.6.2. continue

Оператор «continue» використовується для переривання поточної ітерації циклу. Це корисно, якщо необхідно лишити деякі ітерації і дозволяє уникнути вкладеності коду. Розглянемо обидві ситуації. Припустимо, у нас є цикл, який виводить п'ять послідовних чисел:

```
number = 0
count = 0

while count < 5:
    number += 1
    print(number)
    count += 1
```

Результат роботи нашого коду в консолі:

```
1
2
3
4
5
```

Його можна змінити таким чином, щоб на екран було виведено 5 послідовних парних чисел, використовуючи «continue» за умови, якщо залишок від ділення числа на два не дорівнює 0 (тобто число непарне):

```
number = 0
count = 0
while count < 5:
    number += 1
    if (number % 2) == 1:
        continue
    print(number)
    count += 1
```

Результат роботи нашого коду в консолі:

```
2
4
6
8
10
```

Для розгляду уникнення вкладеності коду припустимо, що необхідно визначити, які з чисел від 0 до 300 поділяються на 3, 3 і 5 або на 3, 5 і 7 одночасно. Тоді код буде виглядати так:

```
number = 0
while number < 300:
    number += 1
    if number % 3 == 0 and number % 5 != 0:
        print(number, "divides by 3")
    elif number % 3 == 0:
        if number % 5 == 0 and number % 7 != 0:
```

```
            print(number, "divides by 3 and 5")
        elif number % 5 == 0 and number % 7 == 0:
            print(number, "divides by 3 and 5 and 7")
```

Тут остання інструкція перебуває на 3-му рівні вкладеності, тобто код вкладений одночасно у 3 конструкції. Використовуючи «continue», можна розділити ці конструкції, закінчуючи ітерацію, якщо умова хибна:

```
number = 0

while number < 300:
    number += 1
    if number % 3 != 0:
        continue
    elif number % 5 != 0:
        print(number, "divides by 3")
    elif number % 7 != 0:
        print(number, "divides by 3 and 5")
    else:
        print(number, "divides by 3 and 5 and 7")
```

Перевага такої конструкції в тому, що вона дозволяє одночасно розділити умови, спростивши їх, і знизити рівень вкладеності коду, тим самим спрощуючи його читання. Як і оператор «break», «continue» завершує ітерацію лише для вкладеного циклу, тоді як ітерація зовнішнього циклу триває.

4.6.3. else

На відміну від умовних конструкцій, у яких «else» виконується тільки якщо умова хибна, «else» в циклах виконується у разі, якщо цикл було успішно завершено.

Наприклад, якщо дитина порахувала від 1 до 5, вона може сказати, що порахувала від 1 до 5:

```
number = 1

while number <= 5:
    print(number)
    number += 1
else:
    print("I've counted from 1 to 5!")
```

Результат роботи нашого коду в консолі:

```
1
2
3
4
5
I've counted from 1 to 5!
```

Не очевидно, навіщо використовувати цей блок якщо «print()» можна розташувати поза тілом циклу і результат буде той самий. Проте, якщо дитина забула кілька цифр, наприклад 4 і 5, вона не зможе порахувати далі і сказати, що порахувала від 1 до 5:

```
number = 1

while number <= 5:
    if number == 4:
        break
    print(number)
    number += 1
else:
    print("I've counted from 1 to 5!")
```

Результат роботи нашого коду в консолі:

```
1
2
3
```

В такому випадку цикл не буде вважатися успішно завершеним і блок «else» не буде виконаний. Крім того, перервати успішне виконання циклу можуть оператори «raise» і «return» (детальніше про «return» ми розглянемо у наступних модулях).

Урок 2
Оператори розгалужень, цикли, виключення