

1. Розгалуження if .. else
2. Логічний тип даних
3. Списки
4. Цикл обходу послідовності for
5. Універсальний цикл while

Логічний тип даних є дуже простим і включає всього 2 можливих значення: True та False -- істина та хиба. Ці значення виникають, насамперед, як результат перевірки (обчислення) будь-яких умов у програмі.

Наприклад, результатом перевірки $10 > 0$ буде True, $91 < 0$ -- False. Коли умова містить змінні, її значення, відповідно, залежить від значень цих змінних на момент перевірки умови (в циклі або розгалуженні).

```
num = raw_input('input N:')
if num == '7':
    print 'Lucky number!'
```

В даному випадку після введення значення користувачем виконується перевірка: якщо введена змінна рівна '7' (нагадую, за умовчанням введені значення вважаються рядками), логічний вираз набуде значення True, умова справдиться і інтерпретатор перейде у вкладений блок коду. Якщо змінна не рівна '7', вираз набуде значення False, тобто "хиба" і умова не справдиться.

Ось це значення умови True або False має свій власний тип даних, що дозволяє нам створювати змінні цього типу і взагалі використовувати його за межами розгалужень та циклів.

Наприклад, ми можемо обчислити логічний вираз завчасно:

```
num = raw_input('input N:')
is_seven = num == '7'
if is_seven:
    print 'Lucky number!'
```

Або (це використовується набагато частіше) створити логічну **змінну-флаг**, яка сигналізуватиме нам про те, чи сталася певна подія. Наприклад, нехай нам необхідно пересвідчитися, чи існують на інтервалі від 1 до n цілі числа, які діляться без остачі одночасно на 11 та 2:

```
n = int(raw_input('input N:'))
flag = False
for i in range(n):
    if (i+1) % 11 == 0:
        if (i+1) % 2 == 0:
            flag = True
print flag
```

ПРОСТІ ТА СКЛАДНІ ЛОГІЧНІ ВИРАЗИ

Автоматично ж значення логічного типу виникають за наявності операцій порівняння у виразі. Для деякої пари значень a та b це:

- $a < b$ -- дорівнює True, якщо a менше b , інакше -- False,
- $a \leq b$ -- дорівнює True, якщо a менше або дорівнює b , інакше -- False,
- $a == b$ -- дорівнює True, якщо a дорівнює b , інакше -- False,
- $a != b$, $a <> b$ -- дорівнює True, якщо a не дорівнює b , інакше -- False,
- $a \geq b$ -- дорівнює True, якщо a більше або дорівнює b , інакше -- False,
- $a > b$ -- дорівнює True, якщо a більше b , інакше -- False.

Вони є *простими* логічними виразами (умовами). В якості a та b можуть використовуватися будь-які змінні, вирази або функції.

На практиці часто доводиться перевіряти складніші випадки, коли мають задовольнятися декілька різних умов. Звичайно, для цього можна використати цілий ряд перевірок -- послідовних, або вкладених одна в одну, як ми це щойно зробили, але такий запис є громіздким. Тому для цього використовуються *складні* логічні вирази, які включають декілька простіших, об'єднаних логічними зв'язками:

- $x \text{ or } y$ -- дорівнює True, якщо хоч 1 з x або y дорівнює True, інакше -- False,
- $x \text{ and } y$ -- дорівнює True, якщо x та y обидва дорівнюють True, інакше -- False,
- $\text{not } x$ -- дорівнює True, якщо x не дорівнює True, інакше -- False.

При цьому x та y можуть бути як простими, так і складними виразами, що дозволяє будувати логічні вирази будь-якої складності. А попередній приклад можна переписати наступним чином:

```
n = int(raw_input('input N:'))
flag = False
for i in range(n):
    if (i+1) % 11 == 0 and (i+1) % 2 == 0:
        flag = True
print flag
```

Логічні операції можуть застосовуватися в одному виразі із математичними. При цьому математичні мають вищий пріоритет виконання, отже будуть обраховані раніше. Далі отримані значення будуть порівняні між собою, а потім результати порівняння "поєднані" логічними зв'язками-операціями.

Першими у виразі обчислюються значення всіх "модифікаторів" not, потім and зліва направо, потім or -- також зліва направо. Так само, як і в арифметичних виразах, круглі дужки змінюють порядок виконання операцій. Але, за наявності в одному виразі різних типів зв'язок, раджу завжди використовувати дужки -- так простіше відстежувати порядок обчислення складних виразів.

Нехай нам необхідно перевірити, чи є заданий рік **високосним**. Для цього він має ділитися націло на 4, але не ділитися на 100, або ділитися на 400:

```
year = int(raw_input('input year:'))
if (year % 4 == 0) and (not(year % 100 == 0) or (year % 400 == 0)):
    print 'is leap'
else:
    print 'is not leap'
```

Втім, якщо відмовитися від "гарної" текстової відповіді, можна прямо вивести значення умови, тобто логічного виразу, який її складає:

```
year = int(raw_input('input year:'))
print (year % 4 == 0) and (not(year % 100 == 0) or (year % 400 == 0))
```

А це вже наstownує на думку, що логічні значення, як рядки і числа, можуть бути конвертовані в інші типи даних та навпаки.

ПЕРЕТВОРЕННЯ ТИПІВ ДАНИХ

True та False можуть бути легко перетворені на рядки. Результатом будуть 'True' та 'False' відповідно.

Також логічні значення перетворюються на числа. Це може бути не так очевидно, але True відповідає одиниці, а False -- нулю. Це працює як для цілих, так і для дійсних чисел:

```
int(True) == 1
int(False) == 0
float(True) == 1.0
float(False) == 0.0
str(True) == 'True'
str(False) == 'False'
```

В зворотньому порядку перетворення працює трохи інакше, але запам'ятати дуже просто: будь-які "непорожні" значення конвертуються в True, будь-які "нульові" -- в False. Для примусового приведення значення до логічного типу використовується вбудована функція bool:

```
bool(None) == False
bool(0) == False
bool(0.0) == False
bool('') == False
```

Всі інші значення відомих нам типів -- істинні:

```
bool(1) == True
bool(10) == True
bool(-1.1) == True
bool('False') == True
bool(-100500) == True
```

Якщо значення інших типів беруть безпосередню участь в логічних операціях (без додаткових операцій порівняння), вони автоматично конвертуються в логічні значення. Отже ми можемо писати

```
if x>0 and y and z:
```

замість

```
if x>0 and bool(y) and bool(z):
```

Наприклад, це дозволяє записати вираз із високосним роком ще наступним чином:

```
year = int(raw_input('input year:'))
print not(year % 4) and ((year % 100) or not(year % 400))
```

Список -- складний тип даних, впорядкована послідовність значень будь-яких типів.

Це значить, що змінна-список містить не одне значення, як число або логічна змінна, а одразу декілька. Причому це можуть бути будь-які значення (рядки, числа, логічні змінні, інші списки) і їх порядок чітко визначений, отже у кожного значення в списку є порядковий номер -- індекс, за яким можна звернутися до окремого елемента списку, щоб прочитати або записати значення. Для визначення списку його елементи беруться в квадратні дужки.

```
example_list = ['one', 1, 2.0, True, [0.1, 0.2, 0.3]]
example_list[0] == 'one'
example_list[1] == 1
example_list[4][1] == 0.2
example_list[5] = '5!' # присвоїли значення новому елементу
```

Найцікавішим є те, що до списків можна застосовувати різні вбудовані функції, призначені для роботи з ними. Наприклад, визначити довжину списку:

```
print len(example_list)
```

ЗРІЗИ

З індексами все досить просто і зрозуміло: якщо значення складається з інших значень, то необхідний якийсь доступ до його складових.

Більш цікавою можливістю списків є взяття "зрізу": крім того, щоб звертатися до окремих елементів, можна вибирати цілі фрагменти послідовності, утворюючи за необхідності нові списки.

- `example_list[i:j]` -- вибере всі елементи списку з *i*-го (включно) по *j*-й (виключаючи),
- `example_list[i:]` -- вибере всі елементи списку з *i*-го (включно) до кінця,
- `example_list[:j]` -- вибере всі елементи списку з початку по *j*-й (виключаючи).

Якщо *i* чи *j* від'ємні, відлік для них буде проводитися з кінця послідовності (до речі, те саме стосується від'ємних індексів). Перевірте, що виведуть `print example_list[-1], example_list[1:4], example_list[-1:13], example_list[4:1], example_list[:-1], example_list[-2:]`.

Також операцію зрізу можна застосовувати і з 3 аргументами:

- `example_list[i:j:k]` -- вибере кожний *k*-й елемент списку з *i*-го (включно) по *j*-й (виключаючи),
- `example_list[i::k]` -- вибере кожний *k*-й елемент списку з *i*-го (включно) до кінця,
- `example_list[:j:k]` -- вибере кожний *k*-й елемент списку з початку по *j*-й (виключно),
- `example_list[::k]` -- вибере кожний *k*-й елемент списку.

k також може бути від'ємним, що призведе до формування нового списку із зворотним порядком елементів. Перевірте самі: `example_list[::-2], example_list[:4:3], example_list[1::-1], example_list[3:0:-1]`.

Є ще одна функція, яка працює схожим чином, -- це вже частково знайома вам `range()`, яка по суті створює список-зріз послідовності цілих чисел:

- `range(i)` -- повертає список з числами від 0 (включно) до *i* (виключаючи, тобто до *i*-1),
- `range(i,j)` -- повертає список з числами від *i* (включно) до *j* (виключаючи, тобто до *j*-1),
- `range(i,j,k)` -- повертає список, що містить кожне *k*-те число від *i* (включно) до *j* (виключаючи, тобто до *j*-1).

Аналогічно, *i*, *j*, *k* можуть бути від'ємними числами.

СПИСОК АРГУМЕНТІВ КОМАНДНОГО РЯДКА. ПОВЕРНЕННЯ

Ви вже знайомі з принаймні одним списком. Це список аргументів `sys.argv`. Він містить всі значення, які були записані в командному рядку для виклику програми, і, як це завжди буває в програмуванні, починає нумерацію своїх елементів з 0.

Під номером (індексом) 0 знаходиться ім'я запущеного файлу. Під номером 1 -- перше значення, під номером 2 -- друге та ін. Аргументи-значення відокремлюються пробілами і складаються інтерпретатором в список `sys.argv` як рядки. Якщо вам необхідно працювати з ними як з даними іншого типу, слід примусово конвертувати тип даних. У випадку коли необхідно передати значення-рядки, що містять пробіли чи інші розділові знаки, які можуть бути сприйняті консоллю або інтерпретатором неадекватно, такі рядки беруться у подвійні лапки.

```
import sys
print sys.argv
```

```
D:\ProgrammingCourse>python test.py 12 12.1 "asda sad as , asda s"
['test.py', '12', '12.1', 'asda sad as , asda s']

D:\ProgrammingCourse>_
```

В тому числі це значить, що зі списком аргументів командного рядка можна робити все те ж, що і з будь-яким іншим списком. Наприклад, обробити всі його аргументи, незалежно від їх кількості:

```
import sys
sum = 0
for arg in sys.argv[1:]: # перебрати всі аргументи, виключивши нульовий
    sum = sum + int(arg) # не забути конвертувати в число для додавання
print sum
```

```
D:\ProgrammingCourse>
D:\ProgrammingCourse>python test.py 1 2 3 4 5
15

D:\ProgrammingCourse>_
```

Або визначити кількість переданих аргументів та вивести повідомлення про помилку у разі їх нестачі:

```
import sys
if len(sys.argv) != 4:
    print 'triangle should have 3 sides, please input 3 numbers'
else:
    print 'OK'
```

```
C:\WINDOWS\system32\cmd.exe

D:\ProgrammingCourse>python test.py 1 2 3 4 5
triangle should have 3 sides, please input 3 numbers

D:\ProgrammingCourse>python test.py 1 2 3 4
triangle should have 3 sides, please input 3 numbers

D:\ProgrammingCourse>python test.py 1 2 3
OK

D:\ProgrammingCourse>python test.py 1 2
triangle should have 3 sides, please input 3 numbers

D:\ProgrammingCourse>python test.py 1
triangle should have 3 sides, please input 3 numbers

D:\ProgrammingCourse>_
```

ЦИКЛ FOR

Поки що він виглядав приблизно так, і це, умовно кажучи, був цикл для того, щоб повторити дію n разів:

```
n = 10
for i in range(n):
    print i
```

Але, після розгляду списків та функції `range()`, стає зрозуміло, що призначення циклу `for` трохи більш загальне -- це обхід послідовностей, наприклад списків. Або рядків. Так, рядки являють собою впорядковані послідовності літер і працюють схожим чином.

Отже нам не обов'язково обмежуватись числами від 0 до N . Ми можемо підготувати заздалегідь будь-який список або рядок і перебрати поелементно. При цьому на кожній ітерації змінна-лічильник циклу набуватиме значення наступного елемента послідовності.

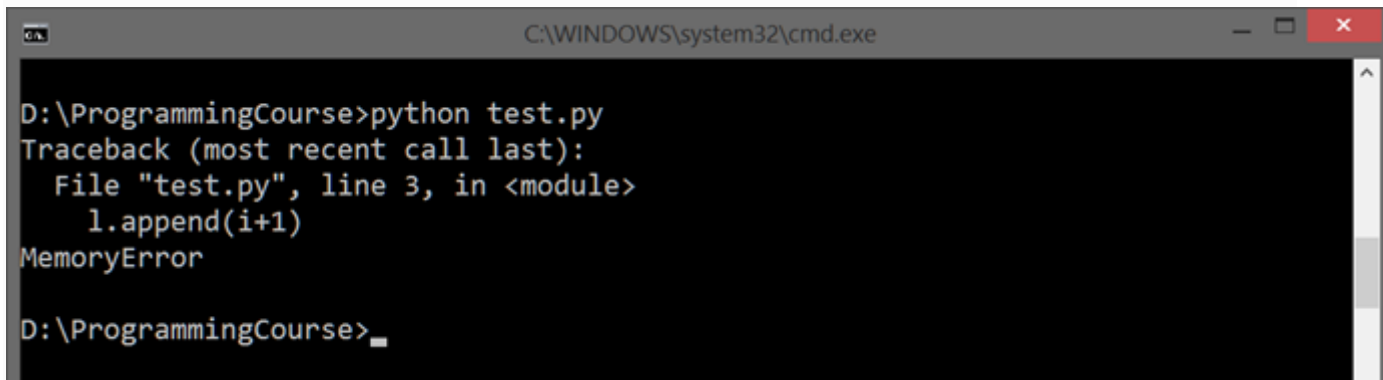
```
# letter послідовно набуває значення кожного символу рядка
for letter in 'qwertyuiopasdfghjkl':
    print letter
list_to_iterate = [1, 2, 3, 4, 8, 2, 43, 12, 31, 20]
# element послідовно набуває значення кожного елемента списку
for element in list_to_iterate:
    print element
# element послідовно набуває значення кожного другого елемента
списку
for element in list_to_iterate[::2]:
    print element
```

Якщо вам необхідно саме перебрати значення послідовності, або ви просто знаєте наперед кількість необхідних ітерацій циклу, цикл `for` підійде

якнайкраще. Крім того, що він призначений саме для цього, його важче впустити у нескінченне повторення, так як будь-яка послідовність має бути скінченною.

Хоча, якщо дуже захотіти, і це, здавалося б, логічне обмеження можна оминати (але зробити це випадково важче ніж для циклу while):

```
l = [0]
for i in l:
    l.append(i+1)
```

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window shows the execution of a Python script "test.py" in the directory "D:\ProgrammingCourse". The output displays a "MemoryError" exception, with a traceback indicating the error occurred at line 3 of the module, specifically at the "l.append(i+1)" statement. The prompt then shows the command "D:\ProgrammingCourse>_" waiting for further input.

```
C:\WINDOWS\system32\cmd.exe

D:\ProgrammingCourse>python test.py
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    l.append(i+1)
MemoryError

D:\ProgrammingCourse>_
```

ЦИКЛ WHILE

Цикл while дозволяє так само перебирати послідовності:

```
n = 10
i = 0
while i < 10:
    print i
    i = i + 1
```

або

```
n = 10
list_to_iterate = [1, 2, 3, 4, 8, 2, 43, 12, 31, 20]
i = 0
while i < len(list_to_iterate):
    print list_to_iterate[i]
    i = i + 1
```

Але він є більш універсальним і може працювати із будь-якою умовою. Коли ви очікуєте якоїсь події, коли вам потрібно повторювати розрахунки до досягнення заданої точності -- в усіх випадках, коли ви не знаєте наперед кількість ітерацій:

```
sum = 0
i = 0
while sum < 100:
    i = i + 1
    sum = sum + i
```


Цикл while продовжує повторювати блок коду, поки значення умови рівне True. І ви завжди повинні бути впевнені, що колись умова змінить своє значення і цикл скінчиться. Бо пропустити збільшення лічильника або просто помилитися в умові набагато простіше ніж "випадково" розтягнути скінченну послідовність до нескінченної.

РОЗГАЛУЖЕННЯ IF .. ELSE

До [розглянутого минулого разу](#) додалася скорочена форма if .. elif .., яка дозволяє побудувати розгалуження більш ніж на 2 гілки, не створюючи для цього додаткових рівнів вкладеності в коді програмі.

```
n = int(raw_input('Input n: '))
if n > 100:
    print 'so large number!'
elif n > 10:
    print 'ok, not bad'
elif n == 0:
    print 'tricky!'
else:
    print 'hey, what are you doing?'
```

Сьогоднішній розбір [логічних виразів](#) в значній мірі також відноситься до розгалужень. І я навіть не знаю, що ще можна додати по цій темі.

len(x) -- повертає довжину послідовності x (списку або рядка).

```
print len('qweqweqweqw qwe qr qr ')
print len([123.1, 32, 66, 23, 'q', 55, 2342])
```

range(x) -- генерує список цілих значень від 0 до x-1.

range(x, y) -- генерує список цілих значень від x до y-1.

range(x, y, z) -- генерує список цілих значень від x до y-1 з кроком z.

```
print range(1, 5, 2) # [1, 3]
print range(10, -5, -3) # [10, 7, 4, 1, -2]
print range(10) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print range(66, 68) # [66, 67]
```

x.append(y) -- додає значення y в кінець списку x

```
x = []
x.append('first!')
print x # ['first!']
```

x.reverse() -- змінює порядок елементів списку x на зворотний

```
x = [1, 2, 3]
x.reverse()
print x # [3, 2, 1]
```

x.upper() -- змінює регістр всіх символів рядка x на верхній.

```
print 'String'.upper() # 'STRING'
```

x.lower() -- змінює регістр всіх символів рядка x на нижній.

```
print 'String'.lower() # 'string'
```

x.replace(substring_old, substring_new) -- замінює всі входження фрагменту substring_old в рядку x на substring_new

x.replace(substring_old, substring_new, count) -- замінює перші count входжень фрагменту substring_old в рядку x на substring_new

```
message = "Good morning, man. What are you doing?"
print message.replace("ng", "n'")
# "Good mornin', man. What are you doin'?"
```

x.find(substring) -- повертає позицію входження (індекс першого символу) фрагменту substring в рядку x або -1, якщо фрагмент не знайдено.

x.find(substring, start_pos) -- повертає позицію входження (індекс першого символу) фрагменту substring в рядку x починаючи з позиції start_pos, або -1, якщо фрагмент не знайдено.

```
str1 = "this is string example....wow!!!"
print str1.find("exam")
print str1.find("exam", 10)
```

ПРИКЛАД НА ЗАКРІПЛЕННЯ

Напишемо програму, яка кодуватиме задане повідомлення [шифром Цезаря](#) із заданим зсувом. Для спрощення проігноруємо регістр символів і приведемо весь текст до нижнього.

```
# імпорт модуля sys для читання аргументів командного рядка
import sys

# збережемо алфавіт мови для пошуку в ньому і підстановки нових літер
alphabet = 'abcdefghijklmnopqrstuvwxyz'
# читаємо текст, який слід закодувати; одразу змінюємо регістр
text = sys.argv[1].lower()
# читаємо зсув для кодування
```

```
shift = int(sys.argv[2])
# оголошуємо змінні для закодованого тексту, закодованої окремої
літери, та номеру літери в алфавіті
coded_text = ''
new_letter = ''
letter_position = None

# пробігаємо весь початковий текст
for letter in text:
    # шукаємо поточний символ в алфавіті
    letter_position = alphabet.find(letter)
    # якщо він там присутній, вираховуємо позицію закодованої літери -
    зі зсувом - і вибираємо літеру алфавіту з новим номером
    if letter_position != -1:
        new_letter = alphabet[(letter_position + shift) %
len(alphabet)]
        # інакше залишаємо символ без змін (пробіли, розділові знаки
та ін.)
    else:
        new_letter = letter
    # додаємо символ до закодованого повідомлення
    coded_text = coded_text + new_letter
# виводимо
print coded_text
```