

1. Прості типи даних
 1. Цілочисельні типи: int, long
 2. Числа із плаваючою крапкою: float
 3. Логічний тип: bool
 4. Рядки: str
2. Незмінювані типи
3. Складні типи або структури даних
 1. Список: list
 2. Тьюпл: tuple
 3. Словник: dict

4. ЦІЛОЧИСЕЛЬНІ ТИПИ: INT, LONG

5. Основним типом для представлення цілих чисел є int. Такий тип існує в будь-якій мові програмування і суттєво відрізняється лише довжиною -- тобто кількістю комірок пам'яті, які відводяться для збереження цілочисельного значення.
6. В python для збереження даних типу int використовується 32 біти (в 32-бітних системах, 64 в 64-бітних; подальші розрахунки приводяться на прикладі 32-бітних), тобто 4 байти ($=32/8$). Ви пам'ятаєте, що 1 біт -- це 1 двійковий розряд, який набуває значення 0 або 1. Маючи 32 біти, ми можемо закодувати $n = 2^{32} = 4\ 294\ 967\ 296$ різних значень. Туди входить 0, від'ємні та додатні числа, отже ціле число типу int може набувати значень у інтервалі $-(n-1)/2 \dots +(n-1)/2$, тобто $-2\ 147\ 483\ 647 \dots 0 \dots 2\ 147\ 483\ 647$.
7. Будь-які значення поза цим інтервалом не можуть бути представлені типом int. У багатьох мовах програмування вихід значень за межі не відстежується і відбувається так зване "переповнення" (при перенесенні розрядів дані починають займати зайві комірки пам'яті, а компілятор/інтерпретатор продовжує працювати зі старими комірками, втрачаючи нові розряди числа), що призводить до неадекватних результатів: додаючи 2 дуже великих числа ви отримуєте дуже мале або від'ємне. Python відстежує такі ситуації і коли результат обчислень виходить за межі int, автоматично конвертує типи даних в інший цілочисельний тип long.

```

>>>
>>> x = 2147483647
>>> type(x)
<type 'int'>
>>> x = x + 1
>>> type(x)
<type 'long'>
>>>

```

8.

9. Всередині інтерпретатора тип long побудований подібно до списків: розряди числа зберігаються як окремі елементи послідовності, а послідовність може мати будь-яку довжину, що дозволяє мати справу із скільки завгодно великими (або малими) цілими числами. Такий формат вимагає програмної реалізації математичних операцій над числами і цей підхід називається “[довгою арифметикою](#)”.

10. Для позначення чисел типу long інтерпретатор використовує літеру L в кінці числа. Це можна спостерігати при виведенні значення в інтерактивному режимі без print’а (використовується інший формат виведення даних). При виведенні на екран print’ом L на кінці буде відсутня. Також ви можете створити значення з L, сигналізуючи інтерпретатору, що повинен використовуватися саме long.

```

>>>
>>> y = 512L
>>> y
512L
>>> print y
512
>>> type(y)
<type 'long'>
>>> _

```

11.

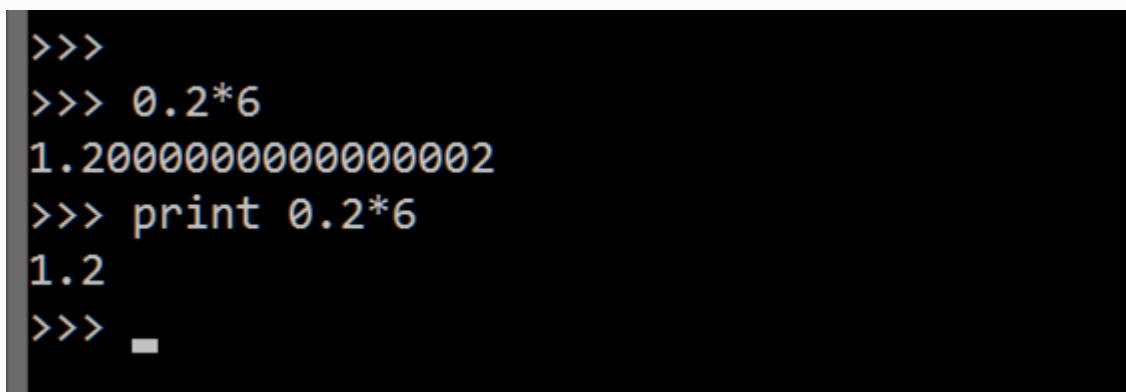
12.

13. ЧИСЛА ІЗ ПЛАВАЮЧОЮ КРАПКОЮ: FLOAT

14. Для того, щоб представити дійсні числа в такому самому форматі, як цілі, довелося б обмежити як цілі, так і дробові їх частини, що суттєво зменшує область їх використання. Тому для дійсних чисел в сучасних комп’ютерах використовується так званий формат “[із плаваючою](#)”

крапкою", який дозволяє зберігати та обробляти як дуже великі, так і дуже малі за модулем числа, фактично обмежуючи при цьому не величину числа, а точність, з якою це число представляється.

15. Для цього дійсні числа зберігаються у формі $x * 2^y$. x називається мантисою ($1 \leq x < 2$, має обмежену кількість знаків), а y -- експонентою ($-1022 \leq y \leq 1022$). Таким чином, числа з плаваючою крапкою все одно мають обмеження, але їх діапазон є надзвичайно великим.
16. Недоліком такого підходу є те, що не всі числа можна точно виразити через ступені двійки. Тому деякі значення (наприклад, $0.1 = 1.999999... \times 2^{-1}$) зберігаються приблизно, а потім округлюються для виведення. Ця похибка накопичується при обчисленнях і може приводити до неочікуваних результатів.



```
>>>
>>> 0.2*6
1.2000000000000002
>>> print 0.2*6
1.2
>>> _
```

- 17.
18. Тому під час проведення обчислень з дійсними числами в програмах завжди повинна закладатися наявність подібної похибки. Наприклад, неможливими є точні порівняння дійсних чисел між собою:
- 19.
- ```
def user_test(a,b):
 return a * b
result_test = user_test(0.2, 6)
result_right = 1.2
if result_test == result_right:
 print 'OK'
else:
 print 'ERROR'
```
20. Даний приклад виведе ERROR, хоча здається очевидним, що  $0.2 * 6 = 1.2$ . Бо при множенні накопичується похибка округлення значення 0.2 і цього достатньо для отримання невірної результату. З урахуванням цієї особливості попередній приклад повинен виглядати так:

- 21.
- ```
def user_test(a,b):
    return a * b
result_test = user_test(0.2, 6)
result_right = 1.2
error = 0.00001 # це значення визначатиме достатню точність обчислень
if abs(result_test - result_right) < error:
    print 'OK'
else:
    print 'ERROR'
```

22. Цікавий факт: тому для представлення сум у серйозних фінансових програмах ніколи не використовуються дійсні числа. Всі обчислення проводяться в цілих числах -- копійках або долях копійок, в залежності від необхідної точності. Наприклад, вартість ноутбука А в правильному інтернет-магазині складає не \$899.99, а 89999 центів. Користувачеві ціна буде виводитися в \$ із роздільником-крапкою, але в усіх розрахунках використовуватиметься цілочисельна. В інакшому випадку суми в рахунках можуть не співпасти із заявленою вартістю товару.
23. Як ви **вже знаєте**, первинним джерелом логічних значень в програмі є порівняння: < <= == != <> => >. У них всіх є цікава особливість: якщо мова йде про числові (int, long, float) або логічні значення, для порівняння їх між собою інтерпретатор автоматично конвертує типи даних.

```
>>>
>>> 1 == 1.0
True
>>> 1L == 1
True
>>> 1 == True
True
>>> 2 < True
False
>>> 2 > True
True
>>>
```

24.

25. Якщо ж нам потрібно точно порівняти два числових значення (включаючи їх типи), для цього можна скористатися операцією **is**:

```
>>>
>>> 1 is 1.0
False
>>> 1L is 1
False
>>> 1 is True
False
>>> 1 is 1
True
>>>
```

26.

27. Ще один цікавий оператор, результатом якого є логічні значення, це **in** -- він перевіряє наявність значення в послідовності. Для словників -- наявність значення серед ключів словника:

```
>>>
>>> d = {1: '1', 2: '2'}
>>> 1 in d
True
>>> 2 in d
True
>>> '1' in d
False
>>>
```

28.

29. Крім того, логічні значення виникають при виконанні логічних операції -- тих самих, що використовуються для поєднання простих умов між собою: and, or, not. Для ілюстрації роботи логічних операцій в математиці часто використовують так звані "таблиці істиності":

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

A	not A
False	True
True	False

30.

31. Звичайно, ви також можете в будь-який момент створювати логічні значення "з нічого" та передавати їх у функції, повертати з функцій або використовувати в обчисленнях:

32.

```
if False and x>0: # ця умова ніколи не справдиться
    print 'OK'
```

З рядками ви також вже багато працювали. Рядок може містити будь-яку послідовність символів, взятих в лапки.

Якщо для позначення рядка застосовано подвійні лапки, всередині можуть вільно використовуватися одинарні, а подвійні інтерпретатор вважатиме кінцем рядка. І навпаки, якщо рядок позначено одинарними, всередині можуть знаходитися подвійні.

Крім того, для позначення рядків можуть використовуватися 3 пари одинарних або подвійних лапок. В такому випадку все, що знаходиться між ними, зберігається "як є":

```
>>>
>>> s = """asd
...     asd\'\'\"
...     """
>>> print s
asd
        asd'"
>>>
```

Рядки є одним із **типів-послідовностей**. Це значить, що до рядків застосовується ряд стандартних функцій: `len()`, `max()`, `min()` та інші. А також, рядок може бути перебраний в циклі `for`. Цікаво, що при цьому елементи, які містяться у рядку, (тобто літери, знаки та ін.) також вважаються рядками, лише одиничної довжини

```
s = 'qwe'
print s[0] # 'q'
print s[0][0] # також 'q'
```

ESCAPE-ПОСЛІДОВНОСТІ

Рядки можуть містити спеціальні символи, які при виведенні рядка обробляються спеціальним чином -- вони називаються `escape`-послідовностями. Найбільш часто використовуються наступні:

- `\n` -- перехід на новий рядок
- `\t` -- вставка табуляції
- `\"` -- подвійні лапки (корисно, якщо необхідно вставити подвійні лапки в рядок, оточений подвійними лапками)
- `\'` -- одинарні лапки (корисно, якщо необхідно вставити одинарні лапки в рядок, оточений одинарними лапками)

Наприклад:

```
>>>
>>> s = 'asd\naa\ta\'\'\"'
>>> print s
asd
aa        a'"
>>>
```

РЕДАГУВАННЯ РЯДКІВ

Також рядки, як і інші прості типи, є **незмінюваними**. Це значить, що ви не можете змінити частину рядка, не створюючи нового. Найпростішим способом оминати це обмеження є використання зрізів, які копіюють послідовність або частину послідовності, до якої застосовуються.

```
s = 'Hello world'
s = s[:6] + 'Python!'
```

Або використання функції `s1.split(separator)`, яка розбиває рядок `s1` на частини, використовуючи переданий рядок `separator` в якості роздільника. Результатом її роботи є список, який складається з цих частин. Ви можете внести в цей список будь-які зміни, а потім об'єднати в новий рядок за допомогою іншої функції -- `s2.join(sequence)`, яка об'єднує послідовність `sequence` (наприклад список рядків), додаючи в якості роздільника між ними рядок `s2`.

```
s = 'Hello world'
s = s.split(' ') # ['Hello', 'world']
s = ', '.join(s) # 'Hello, world'
```

ФОРМАТУВАННЯ РЯДКІВ

Ще однією можливістю рядків є "форматування" -- можливість вставки в рядок даних у необхідному форматі без додаткової конкатенації. Для цього використовується оператор `%`, першим операндом якого є рядок-шаблон, а другим -- тьюпл із даними для вставки.

Рядок-шаблон містить "заглушки", які позначають, в які місця повинні вставлятися дані та в якому форматі. Наприклад:

```
s = 'My name is %s, I'm %s years old.' % ('Vasia', 21)
```

аналогічне запису

```
s = 'My name is ' + 'Vasia' + ', I'm ' + str(21) + ' years old.'
```

Дані вставляються в рядок в тому порядку, в якому вони знаходяться в тьюплі. "Зاغлушка" `%s` позначає вставку даних з приведенням їх до типу рядок-- `str()`.

Іншими "заглушками" є

- `%d` -- ціле десяткове число,

- %x, %X -- ціле шістнадцяткове число,
- %f -- дійсне число в десятковому представленні
- %e, %E -- дійсне число в експонентному представленні
- %% -- знак відсотку

Для форматування числових значень всередині "заглушок" (між % та літерою) можуть використовуватися додаткові параметри:

- + -- відображати знак числа,
- Пробіл -- додати пробіл перед додатним числом замість "+",
- m.n -- вставити число як рядок з m символами, залишити n знаків після коми; при цьому рядок буде розширено до необхідної довжини пробілами,
- 0m.n -- те саме, але рядок буде розширено до необхідної довжини нулями.

Наприклад:

```
>>>
>>> print "%010.2f" % (123)
0000123.00
>>> print "%+010.2f" % (123)
+000123.00
>>> print "% 010.2f" % (123)
 000123.00
>>> print "% 010.0f" % (123)
 000000123
>>> print "%10.0f" % (123)
      123
>>>
```

Про списки ви також знаєте вже досить багато. Із нового -- це єдиний змінюваний тип даних із тих, що ми розглядали раніше. Це значить, що для внесення змін в список не потрібно створювати новий, а можна змінити будь-який окремий його елемент.

Тьюпли (або кортежі) нічим принципово не відрізняються від списків, але задаються в круглих дужках і є незмінюваними. Найчастіше вони використовуються для передачі даних між функціями, коли необхідно

захистити передані значення від випадкового перезапису, або просто для виведення структурованих даних, які не потрібно обробляти далі.

Наступна програма моделює того самого пасажирів, який їздить в автобусі і чекає на квиток із "щасливим" номером. Перша функція генерує випадковий номер квитка, друга -- перевіряє, чи є квиток щасливим. Третя функція моделює отримання квитка пасажиром і повертає тьюпл із двома значеннями: номером отриманого квитка та логічним значенням в залежності від того, чи є номер щасливим.

```
from random import randint

def generate_ticket_number():
    return randint(0, 999999)

def is_ticket_lucky(num):
    num = str(num)
    num = '0' * (6 - len(num)) + num
    return int(num[0]) + int(num[1]) + int(num[2]) == int(num[3]) + int(num[4]) + int(num[5])

def get_ticket():
    number = generate_ticket_number()
    is_valid = is_ticket_lucky(number)
    return (number, is_valid)

my_ticket_in_bus = get_ticket()
if my_ticket_in_bus[1] == True:
    template = 'Woohoo! I\'m lucky, my ticket is %s'
else:
    template = 'My ticket is %s. Will try tomorrow.'
print template % (my_ticket_in_bus[0])
```

ДВОВИМІРНІ СТРУКТУРИ ДАНИХ

Не сказати, що це якась особлива можливість списків або тьюплів, але досить популярний варіант використання.

Самі по собі структури даних є одновимірними, тобто представляють послідовність значень, які слідує одне за одним. Але, вкладаючи їх одна в одну, ми можемо моделювати дво- або багатовимірні речі. Наприклад, таблиці або математичні [матриці](#). Задачі на їх обробку є досить розповсюдженими в програмуванні -- в першу чергу, це будь-яка робота із графікою (як із звичайними зображеннями, так і з тривимірною).

Матриця являє собою прямокутну таблицю, в комірках якої записані числа. В таблиці можна виділити рядки та стовпці, відповідно кожний елемент матриці має адресу, яка записується 2 числами -- номерами стовпця та

рядка, до яких належить елемент. Іншу пара чисел -- кількості стовпців та рядків -- називають розмірністю матриці.

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 7 \\ 4 & 9 & 2 \\ 6 & 2 & 5 \end{bmatrix}$$

Таку матрицю легко представити у вигляді вкладених списків:

```
matrix = [[1, 2, 3], [1, 2, 7], [4, 9, 2], [6, 2, 5]]
```

Далі, коли ми звернемося за індексом, наприклад, 0, отримаємо список, який містить нульовий рядок матриці:

```
print matrix[0] # [1, 2, 3]
```

Вказавши одразу 2 індекси, ми одразу звернемося до цього вкладеного списку і отримаємо окремий елемент матриці:

```
print matrix[0][1] # 2
```

На жаль, така вкладена структура не гарантує того, що дані є коректними. Наприклад, зовнішній список може містити вкладені списки різної довжини, які не можуть представляти прямокутну числову матрицю:

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

Втім, перевірити, чи є структура коректною нескладно, для цього можна обійти вкладені списки і порівняти їх довжину, пересвідчуючись при цьому, що всі структури є списками:

```
def is_matrix(mat):
    if not isinstance(mat, list) or not isinstance(mat[0], list):
        return False
    size = len(mat[0])
    for row in mat:
        if not isinstance(row, list) or len(row) <> size:
            return False
    return True
```

За необхідності ми можемо обійти всі елементи матриці, використовуючи подвійний цикл. Наприклад, вивести всі елементи матриці:

```
def print_elements(mat):
    for i in range(len(mat)):
        for j in range(len(mat[i])):
            print mat[i][j]
```

Або зручно вивести матрицю у вигляді прямокутника:

```
def output_matrix(mat):
    for row in mat:
        string = ''
        for el in row:
            string = string + "%4d" % (el)
        print string
```

Напишемо програму, яка *транспонує* задану матрицю, тобто міняє місцями її стовпці та рядки. Для цього необхідно створити новий список (змінюючи старий, ми можемо втратити якісь елементи і результат буде невірним), в ньому створити необхідну кількість вкладених списків та заповнити їх елементами, відповідно змінюючи їх індекси (`matrix2[i][j] = matrix[j][i]`). Для цього використаємо попередньо визначені функції `is_matrix()` та `output_matrix()`:

```
matrix = [[1,2,3],[1,2,7],[4,9,2],[6,2,5]]
```

```
def transpone(mat):
    mat2 = []
    for i in range(len(mat[0])):
        for j in range(len(mat)):
            if j==0:
                mat2.append([])
            mat2[i].append(mat[j][i])
    return mat2
```

```
if is_matrix(matrix):
    output(matrix)
    print "transponed:"
    output(transpone(matrix))
```

Принципово новим для вас типом даних є словники. А вони є однією з найбільш гнучких готових структур у python.

Як і списки та тьюпли, словники можуть містити елементи із даними будь-яких типів. Принципова відмінність полягає в тому, що ці елементи не мають строго визначеного порядку і доступ до них надається не по номерах-індексах, а за допомогою ключів. Тобто словник складається із пар "ключ"- "значення", де ключі мають відноситися до незмінних типів (числа, рядки, тьюпи), а "значення", що їм відповідають можуть бути будь-якими (включаючи списки або інші словники).

```
d = {(1,2): 'tuple', 1: 'int', 2.0: 'float', '3': 'string'}
print d[(1,2)] # 'tuple'
print d[1] # 'int'
print d[1.0] # 'int', як і при порівнянні в ключах не розрізняються різні
числові типи
print d[2.0] # 'float'
print d['3'] # 'string'
```

Як і списки, словники є змінюваним типом даних. Це значить, що в будь-який момент ми можемо додати до словника нові елементи:

```
d['qwe'] = 'asd'
```

Для видалення значень із словника використовується функція `d.pop(key)`, яка видаляє із словника `d` значення, що відповідає ключу `key`.

```
d.pop((1,2))
print d # {1: 'int', 2.0: 'float', '3': 'string', 'qwe': 'asd'}
```

Так як словник не є послідовністю, його елементи зберігаються не в тому ж порядку, в якому вони були додані в словник. При виведенні словника його елементи за умовчанням сортуються за значеннями ключів. При спробі перебрати словник напряду в циклі, також використовується список ключів словника.

Словники є зручним способом представлення структурованих даних, таких як записи в телефонному довіднику, дані про товари у магазині, бібліографічні дані книг та ін.

```
people = {'Alice': {'phone': '2341', 'addr': 'Foo drive 23' },
          'Beth':  {'phone': '9102', 'addr': 'Bar street 42'}}
```

```
name = 'Alice'
key = 'phone'
if name in people:
    print "%s phone is %s" % (name, people[name][key])
```

Також словники є дуже зручними для кодування. Розглянемо наш [старий приклад із шифром Цезаря](#). У випадку, якщо нам часто треба кодувати довгі тексти, було б ефективніше побудувати словник, в якому б зберігалися перетворення всіх літер і використовувати його замість перерахунку кожної літери окремо (ну ок, шифр Цезаря надто простий, різниця буде непомітна, але для більш складних шифрів така підготовка може суттєво оптимізувати програму). Саме для цього і зручно використати словник:

```
import sys
alphabet = 'abcdefghijklmnopqrstuvwxyz'
text = sys.argv[1].lower()
shift = int(sys.argv[2])
coded_text = ''
new_letter = ''
letter position = None

coder = {}
for i in range(len(alphabet)):
    coder[alphabet[i]] = alphabet[(i + shift) % len(alphabet)]

for letter in text:
    if letter in coder:
        coded_text = coded_text + coder[letter]
    else:
        coded_text = coded_text + letter
print coded_text
```

СТАНДАРТНІ ФУНКЦІЇ ДЛЯ СЛОВНИКІВ

d.clear() -- видаляє всі значення із словника d

```
d = {1:10, 2:20, 3:30}
d.clear()
print d # {}
```

d.copy() -- повертає копію словника d

```
d1 = {1:10, 2:20, 3:30}
d2 = d1
d3 = d1.copy()
d1[1] = '!'
print d2[1] # '!'
print d3[1] # 10
```

d.get(key, default) -- повертає значення, що відповідає ключу key словника d; якщо ключ відсутній, повертає значення default

```
d = {1:10, 2:20, 3:30}
print d.get(1, 'Nothing found') # 10
print d.get(0, 'Nothing found') # 'Nothing found'
```

d.keys() -- повертає список ключів словника d

```
d = {1:10, 2:20, 3:30}
print d.keys() # [1, 2, 3]
```

d.values() -- повертає список значень словника d

```
d = {1:10, 2:20, 3:30}
print d.values() # [10, 20, 30]
```

d.items() -- повертає список тьуплів, кожен з яких містить пару (ключ, значення) для словника d

```
d = {1:10, 2:20, 3:30}
print d.items() # [(1, 10), (2, 20), (3, 30)]
```