1. Концепція модульності
2. Функції
    1. Оголошення та використання функцій
    2. Області видимості змінних
    3. Рекурсія
3. Модулі
4. Пакети модулів

Отже, ми можемо взяти будь-яку частину коду, дати їй ім’я і винести з основного потоку програми. Наприклад, програма, яка шукає на відрізку від 0 до 99 всі числа, записані однаковими цифрами, може бути записана так:

```
n = 100
for x in range(n):
    is_cool = x < 10 or (x / 10 == x % 10)
    print x, is_cool
```

Або так:

```
def check_if_is_cool(number):
    is_cool = number < 10 or (number / 10 == number % 10)
    print number, is cool

n = 100
for x in range(n):
    check_if_is_cool(x)
```

Або взагалі так. І це буде найправильніший варіант, так як виведення даних має відношення до самої програми і залишається в циклі, а в окрему функцію виноситься лише перевірка, яка сама по собі може знадобитися і в іншому місці:

```
def is_cool(number): # limited by 0 <= number <= 99
    return number < 10 or (int(number) / 10 == number % 10)

n = 100
for x in range(n):
    print x, is_cool(x)
```

Функція задається за допомогою ключового слова `def`, за яким слідує назва функції (за цією назвою ми далі можемо її викликати) та круглі дужки із переліком аргументів функції. Наступна двокрапка вказує інтерпретатору чекати на вкладений блок коду (як і у випадку з алгоритмічними конструкціями), далі слідує сам блок -- тіло функції, виділене відступами, яке, звичайно, може містити будь-які конструкції.

Аргументи функції -- це дані, які передаються у функцію при її виклику і мають нею оброблятися для отримання результату. Сам отриманий результат повертається за допомогою оператора return:

```
def is_repeat(string):
    result = string == string[0] * len(string)
    return result
```

При поверненні результату робота функції припиняється. Тобто приклад із попередньої лекції:

```
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
```

можна переписати наступним чином:

```
def dividable_11_2(x):
    return x % 11 == 0 and x % 2 == 0

def has_dividable_11_2_before_n(n):
    for i in range(n):
        if dividable_11_2(i + 1):
            return True
    return False

n = int(raw_input('input N:'))
print has_dividable_11_2_before_n(n)
```

Тоді, щойно умова справдиться, функція поверне результат і закінчить роботу. Якщо цикл закінчиться і результат досі не повернено, це означає, що умова не справдилася для жодного з чисел на інтервалі і слід повернути False.

АРГУМЕНТИ ФУНКЦІЇ

Функції можуть приймати 0, 1, 2 або скільки завгодно аргументів. При виконанні інтерпретатор створить для них одноіменні змінні, які будуть доступні функції під час її виконання, та ініціалізує їх переданими значеннями. Якщо аргументів функції декілька, вони сприймаються саме в тому порядку, в якому записані в оголошенні функції.

Аргументи, які записуються при оголошенні просто через кому, є **обов'язковими**. Тобто, якщо функція оголошена з 1 аргументом, а ми спробуємо викликати її, не передавши значень, отримаємо повідомлення про помилку.

```
D:\ProgrammingCourse>python test.py
input N:12
Traceback (most recent call last):
  File "test.py", line 44, in <module>
    print has_dividable_11_2_before_n()
TypeError: has_dividable_11_2_before_n() takes exactly 1 argument (0 given)
```

Але часто буває корисно мати якісь **значення за умовчанням**, щоб навіть, якщо у функцію не передані необхідні дані, вона все одно вірно відпрацьовувала. Наприклад, обчислюючи результат для найпоширенішого варіанту. Такі значення задаються прямо в переліку аргументів при оголошенні функції:

```
def has_dividable_11_2_before_n(n = 99):
```

Відповідно, функція може бути викликана без передачі таких аргументів. Тобто вони є **необов'язковими**.

```
print has_dividable_11_2_before_n()
```

Перепишемо попередню програму так, щоб вона шукала числа на проміжку, які діляться без остачі на будь-які задані числа:

```
def is_dividable(x, dividers = [11, 2]):
    for divider in dividers:
        if x % divider != 0:
            return False
    return True

def has_dividable_numbers(n = 99, dividers = [11, 2]):
    for i in range(n):
        if is_dividable(i+1, dividers):
            return True
    return False

n = int(raw_input('input N:'))
print "11, 2 : " + str(has_dividable_numbers(n))
print "5 : " + str(has_dividable_numbers(n, [5]))
print "2, 3, 5 : " + str(has_dividable_numbers(n, [2, 3, 5]))
```

Якщо функція передбачає обидва типи аргументів, обов'язкові записуються першими (як при оголошенні, як і при виклику); всі передані значення передаються в аргументи зліва направо по одному. Наприклад, маємо функцію з 3 обов'язковими аргументами і 3 необов'язковими:

```
def multiple_args_function(arg1, arg2, arg3, arg4 = 0, arg5 = 1, arg6 = 2):
    print arg1, arg2, arg3, arg4, arg5, arg6
```

- При виклику multiple_args_function(1) -- отримаємо помилку, так як 1 підставляється в arg1, а значення інших обов'язкових аргументів залишаються невизначеними.

- При виклику multiple_args_function(1, 2, 3) -- 1, 2 та 3 будуть підставлені як значення 3 обов’язкових аргументів, для необов’язкових будуть взяті значення за умовчанням.
- При виклику multiple_args_function(1, 2, 3, 4) -- 1, 2 та 3 будуть підставлені в обов’язкові аргументи, 4 -- в 1-й необов’язковий arg4.
- І т.д.

Якщо нам необхідно передати аргументи в іншому порядку, ми можемо при виклику функції передати значення разом із іменами аргументів:

multiple_args_function(1, 2, 3, arg5=4) -- 1, 2 та 3 будуть підставлені в обов’язкові аргументи, 4 -- в arg5.

За відповідністю типів даних в аргументах має стежити сам програміст, Python 2 відстежує лише кількість та порядок аргументів і, якщо при виклику функції були передані некоректні значення, такий випадок має перевірятися функцією, або призведе до помилок виконання (нехай ми викликаємо функцію has_dividable_numbers(n, 9)):

```
Traceback (most recent call last):
  File "test.py", line 63, in <module>
    print "9 : " + str(has_dividable_numbers(n, 9))
  File "test.py", line 55, in has_dividable_numbers
    if is_dividable(i+1, dividers):
  File "test.py", line 48, in is_dividable
    for divider in dividers:
TypeError: 'int' object is not iterable
```

У багатьох інших мовах програмування можливість вказати типи аргументів при оголошенні функції є. Наскільки мені відомо, починаючи з версії інтерпретатора 3.0, вона з’явилася і в Python.

ОБЛАСТІ ВИДИМОСТІ ЗМІННИХ

Як уже зазначалося у відео-лекції, весь простір програми розділено на області видимості, в кожній з яких доступний свій набір змінних.

Код основної програми визначає **глобальну область видимості**. Всі сутності з неї (функції та змінні) доступні скрізь в межах програми, включаючи вкладені області.

Кожна оголошена функція створює власну -- **локальну область видимості**. В ній можуть створюватися локальні імена (змінні та функції), які будуть доступні в межах даної функції та у вкладених областях видимості (наприклад, якщо всередині функції оголошені інші вкладені в

неї функції). Крім того, з локальних областей видимості можна прочитати значення із зовнішніх, але не можна їх змінити.

В локальній області видимості можуть вільно створюватися змінні та функції з такими ж іменами, які існують в інших областях видимості. При цьому вони “перекриватимуть” імена із зовнішніх областей.

Замість довгих пояснень, простіше прокоментувати приклад з відео:

```
# головна програма -- глобальна область видимості A
a = 1                           # глобальна змінна A.a
b = 2                           # глобальна змінна A.b

def myfunction1(arg_a, arg_b):  # ім’я A.myfunction1 знаходиться
                                # у глобальній області видимості,
                                # всередині можна звертатися до будь-яких
                                # функцій та # змінних A.*,
                                # але вони не можуть бути змінені.
                                # Аргументи функції зберігаються
                                # як локальні змінні A.arg_a та A.arg_b

    arg_a = 'one'               # локальна змінна A.myfunction1.arg_a,
                                # доступна лише всередині myfunction1 та
                                # у вкладеній функції inner_function
    c = 'three'                 # інша локальна змінна A.myfunction1.c

    def inner_function(name):   # вкладена функція створює нову вкладену
                                # область видимості
                                # A.myfunction1.inner_function із локальною
                    # змінною A.myfunction1.inner_function.name,
                    # до якої має повний доступ

        inner_a = 1             # локальна змінна
                                # A.myfunction1.inner function.inner a
        return 'Hello ' + str(name) + '!'

    print '   myfunction1 scope:'
    print inner_a               # myfunction1 нічого не знає про вкладену
                                # область видимості -- помилка
    print arg_a                 # локальна змінна -- є доступ
    print arg b                 # локальна змінна -- є доступ
    print c                     # локальна змінна -- є доступ
    print b                     # глобальна змінна (A.b) -- є доступ
                                # для читання
    print inner_function(c)     # локальна функція -- є доступ
    print 'and NOW I\'m calling the myfunction2!'
    myfunction2(arg_a, arg_b)   # myfunction2 знаходиться на тому ж
                                # рівні (визначена безпосередньо
                                # в області A), отже може бути викликана
                                # з сусідньої функції

def myfunction2(arg_a, arg_b):  # створює область видимості A.myfunction2
                                # із 2 локальними змінними --
                                # аргументами функції
    arg_a = 'ONE'               # локальна змінна A.myfunction2.arg_a
    print '   myfunction2 scope:'
    print arg_a                 # локальна змінна -- є доступ
```

```
    print arg_b                 # локальна змінна -- є доступ

# повернулися в глобальну область видимості A
myfunction1(a,b)                # функція та змінні знаходяться в поточній
                                # області видимості -- є доступ
myfunction2(a,b)                # функція та змінні знаходяться в поточній
                                # області видимості -- є доступ
print '   main program scope:'
print a                         # змінна в поточній області видимості -- є
доступ
print b                         # змінна в поточній області видимості -- є
доступ
print c                         # змінна з області видимості A.myfunction1 --
                                # немає доступу, помилка
print inner_function('Ivanko'); # функція з області видимості A.myfunction1 --
                                # немає доступу, помилка
```

Рекурсія -- цікавий прийом в програмуванні, коли функція викликає сама себе.

```
def factorial(i):
    if i == 0:
        return 1
    else:
        return i * factorial(i - 1)
```

Це зручно у випадках, коли ми можемо розбити розв’язувану складну задачу на одну або декілька більш простих, а потім зібрати результати розв’язання менших задач в 1 результат для складнішої.

У випадку з факторіалом ідея полягає в наступному:

- Ми знаємо, що 0! = 1 -- це найпростіший випадок, для якого задача розв’язується елементарно.
- Ми знаємо, що факторіал будь-якого числа легко розрахувати, знаючи факторіал попереднього: 1! = 0! * 1, 2! = 1! * 2, n! = (n-1)! * n -- отже для того, щоб обчислити n! достатньо знати значення попереднього факторіалу, для (n-1)! -- ще попереднього і т.д.
- Отже ми змушуємо нашу функцію обчислювати більш прості факторіали раз за разом, поки задача не зведеться до елементарного 0!, для якого ми знаємо точну відповідь. На кожному рівні заглиблення функція повертає обчислене значення (i-1)! і, “повертаючись назад”, ми можемо просто домножити його на i, щоб отримати розв’язок більш складної задачі i!

Тобто:

```
def factorial(i):
    if i == 0:
        return 1                # для i=0 розв'язком задачі буде 1
                                # else не обов'язкове: якщо умова
```

```
справдилася,
                                  # функція вже завершилася
    return i * factorial(i - 1) # інакше розв'язуємо простішу
задачу
                                  # і будуємо розв'язок цієї,
відштовхуючись від неї
```

Іншим класичним прикладом використання рекурсії для розв’язання задачі є гра “ханойські вежі”, яка розв’язується саме рекурсивно.

В загальному випадку використання рекурсії є неефективним, так як при кожному виклику створюється нова область видимості та комп’ютером виділяється окрема область пам’яті для виконання функції. При цьому попередньо виділена пам’ять також лишається задіяною, очікуючи на розв’язок вкладеної задачі. Всі обслуговуючі процеси займають час та потребують багато ресурсів комп’ютера, тому розв’язок задач з великою вкладеністю за допомогою рекурсії часто є неможливим суто по технічних причинах.

З іншого боку, в основі рекурсії завжди лежить цикл. Тому будь-яка рекурсія може бути замінена одним або кількома циклами без необхідності виділення такої кількості ресурсів. На жаль, розв’язок задачі без використання рекурсії часто є, м’яко кажучи, неочевидним.

Це, в тому числі, стосується задач обробки вкладених структур. Наприклад, припустимо, що у нас є список із числами та іншими вкладеними списками, і нам необхідно обчислити суму всіх чисел, які знаходяться всередині структури. Якби ми точно знали рівень вкладеності, задачу було б легко розв’язати відповідною кількістю вкладених циклів. Але структура може бути довільною, тому найпростішим є скласти всі елементи зовнішнього списку, виділити з нього вкладені списки і розв’язати для них таку саму задачу, додавши потім їх суми до суми елементів поточного списку.

```
def list_sum(el):             # шукаємо суму елементів el
    if isinstance(el, list):  # якщо el список, для цього
необхідно скласти всі його елементи
        sum = 0
        for item in el:
            sum = sum + list_sum(item) # додаємо до суми el суму
кожного його елемента,
                                       # а наступний list_sum
нехай сам вирішує, як її рахувати
        return sum            # повертаємо суму для списку
    else:                     # інакше el число і нема що складати
-- його і повертаємо
        return el

print list_sum([1, 1, -1, [2, -2, [[4, -4, [777]], 3, [5, -5], -
3]], -1])
```

Пам’ятайте: так як рекурсія по суті також є циклом, але має складніший запис, заблукати в ній, забувши про умову виходу набагато простіше ніж у циклі!

Якщо код програми є складним, розбиття її на окремі функції допомагає спростити його для візуального сприйняття. Якщо цього недостатньо, є сенс винести частину функцій та пов’язаних з ними оголошень за межі основного файлу програми.

Такі додаткові файли з кодом, що використовується в програмі, називаються модулями. Найчастіше вони містять оголошення функцій та констант, які далі можуть бути підключені (імпортовані) в головну програму і вільно в ній використовуватися.

Наприклад, ми можемо створити файл prometheus_math.py і оголосити в ньому функції для обчислення факторіалу, чисел Фібоначчі та інших розрахунків, що ми використовували протягом курсу. Далі ми можемо створити нову програму і підключити до неї модуль prometheus_math (він має знаходитися в одній папці з програмою, ім’я модуля збігається з іменем файлу, лише без розширення .py) і використовувати всі ці функції в новій програмі.

Як бачите, ще однією перевагою модулів є те, що оголошення, зроблені в модулі, можуть використовуватися не в одній, а в будь-якій програмі, яку ви писатимете далі.

Аналогічним чином працює вбудований модуль math, який ми вже використовували на початку курсу. Він містить константи math.e та math.pi, що дозволяє вам звертатися до готових значень замість повторного визначення їх в своєму коді. В ньому ж знаходиться ряд математичних функцій, які ви так само можете використовувати, не замислюючись про їх технічну реалізацію.

ІМПОРТ З МОДУЛІВ ТА ЙОГО ВИДИ

Все, що необхідно, це підключити модуль до своєї програми:

```
import math
```

і використовувати його функції та константи, звертаючись до них через ім’я модуля з крапкою:

```
print math.pi
```

Запис math.pi означає, що значення pi знаходиться в просторі імен модуля math. Це дуже схоже з областями видимості функцій: кожен модуль має власний простір імен -- це необхідно для того, щоб модуль мав власний

набір змінних та функцій і жодним чином не залежав від інших модулів, які ви підключите.

Наприклад, в своєму модулі prometheus_math ви підключили модуль math, щоб взяти з нього значення математичної константи e (звичайно, модулі можна підключати як до самої програми, так і до інших модулів). А потім визначили в ньому власну функцію exp():

```
def exp(x):
    return math.e ** x
```

Якщо ви підключите до основної програми і math, і prometheus_math, у вас буде дві функції з однаковими іменами. Але це не призведе до помилки, так як вони визначені в різних просторах імен: prometheus_math.exp та math.exp і при зверненні до них інтерпретатор чітко знатиме, яку саме ви маєте на увазі.

Також ви можете імпортувати функції будь-якого модуля одразу в простір імен вашої програми (або модуля):

```
from math import *
print pow(10, 2)
```

В такому випадку всі оголошення модуля будуть скопійовані в поточний простір імен. Але з’являється ризик того, що у вашому просторі імен виникнуть однакові імена і в такому випадку ваша програма може повести себе не так як ви очікуєте.

Для того, щоб уникнути плутанини, з підключених модулів часто імпортують не всі оголошення, а лише необхідні. При цьому перелічені імена (разом із значеннями або функціями, що за ними стоять) будуть скопійовані в поточний простір імен, а всі інші залишаться недоступними:

```
from math import exp, pi, pow
print pow(10, 2)
print e  # помилка
print math.e  # помилка
```

Якщо вам необхідно імпортувати одноіменні функції з кількох модулів, для них можна задати псевдоніми. Тоді функція буде теж скопійована в поточний простір імен, але під іншою назвою:

```
from math import exp, pi, pow as math_power
from prometheus_math import exp as my_exp
print exp(10) # викличе exp з модуля math
print my_exp(10) # викличе exp з модуля prometheus_math
print math_power(10, 2) # викличе pow з модуля math
```

Зверніть увагу: для підключення інтерпретатор шукає модулі у поточній папці. Якщо ви підключаєте модуль всередині файлу (програми або іншого

модулю) -- це папка, в якій знаходиться цей файл. Якщо ви підключаєте модуль в інтерактивному режимі інтерпретатора -- це папка, в якій його було запущено. Якщо в поточній папці файл модуля не знайдено, інтерпретатор перевіряє деякий шлях, вказаний в налаштуваннях Python (відрізняється для різних операційних систем). Саме там знаходяться більшість стандартних модулів, туди ж ви можете додавати власні модулі, туди ж можна додавати модулі, знайдені в інтернеті, для використання в своїх програмах.

ІНІЦІАЛІЗАЦІЯ МОДУЛІВ

Крім оголошення функцій та значень модуль може містити виконуваний код. Такий код буде виконано в момент підключення модулю і він використовується для підготовки модулю для роботи.

Наприклад, вже знайомий вам модуль sys, який ми використовували для отримання аргументів, переданих програмі. Значення sys.argv, в якому зберігаються ці аргументи, -- не функція і не константа, а отже не може бути ініціалізоване заздалегідь. Це може бути прикладом ініціалізації модуля: при його підключенні виконується деякий код, який звертається до середовища, в якому запущено програму, отримує звідти параметри її запуску і зберігає в змінну argv модуля, роблячи їх таким чином доступними в нашій програмі.

Також це значить, що Python не робить принципової відмінності між модулем і програмою. Будь-яка ваша програма може бути імпортована як модуль -- при цьому її буде виконано, що не завжди доцільно. Будь-який модуль може бути запущено як програму -- правда, багато з них містять лише оголошення тому видимого результату роботи не буде. Таке розділення є скоріш умовним і робиться програмістом для зручності побудови програм.

P.S. Я знаю, що math і sys є обгортками над функціями C і побудовані дещо інакше, але загалом принцип роботи ілюструють, тому використані тут саме як вже знайомий вам приклад.

ПАКЕТИ

Коли модулів стає забагато, виникає необхідність групувати їх далі. Для цього файли модулів розкладаються по папках.

Ви знаєте, що інтерпретатор шукає модулі в поточній папці та у спеціально призначеному для цього місці, отже необхідно якимось чином показати йому, що папка поряд з вашою програмою -- не просто папка з файлами, а

містить модулі для підключення. Для цього в папці повинен знаходитися файл __init__.py -- він може бути порожнім, але сама його наявність сигналізує інтерпретатору, що папка із ним є пакетом модулів і може використовуватися в програмі.

Як і модулі, пакети створюють нові простори імен:

```
import my_prometheus_package.prometheus_math # модуль
prometheus_math шукатиметься
                                                # в пакеті
my_prometheus_package (в одноіменній папці)
print my_prometheus_package.prometheus_math.exp(1)
```

Все, що я розповідав про види імпорту поширюється також на пакети. Лише в іменах додається додатковий елемент через крапку -- назва пакету. Пакети можуть вкладатися в інші пакети, аналогічно додаючи нові простори імен. Подальше структурування програми обмежується лише вашою фантазією.

Як і модулі, пакети можуть містити код, який буде виконано під час ініціалізації пакету, -- він записується в самому файлі __init__.py

isinstance(x, y) -- перевіряє, чи є x значенням типу y.

```
print isinstance('string', str) # True
a = ['q']
print isinstance(a, str) # False
print isinstance(a, list) # True
```

sorted(x) -- повертає список - відсортовану послідовність (рядок або список), яка складається з елементів x.

```
print sorted('abdgc') # ['a', 'b', 'c', 'd', 'g']
print sorted([3, 2, 1]) # [1, 2, 3]
```

ФУНКЦІЇ СПИСКІВ

x.sort() -- сортує елементи списку x за зростанням.

```
x = [7, 5, 3, 1]
x.sort()
print x # [1, 3, 5, 7]
```

x.extend(y) -- додає елементи списку y в кінець списку x.

```
x = [7, 5, 3, 1]
x.extend([1, 2])
print x # [7, 5, 3, 1, 1, 2]
```

x.insert(i, y) -- вставляє елемент y на позицію i в списку x, при цьому існуючі елементи списку зсуваються.

```
x = [7, 5, 3, 1]
x.insert(1, 0)
print x # [7, 0, 5, 3, 1]
```

ФУНКЦІЇ МОДУЛЮ RANDOM

random.randint(a, b) -- повертає випадкове ціле число на відрізку від a до b включно.

```
print random.randint(10, 20)
print random.randint(1, 100500)
```

random.choice(x) -- повертає випадковий елемент з непорожньої послідовності (списку або рядка) x.

```
print random.choice(['back', 'forward', 'left', 'right'])
print random.choice('abcdefghijklmnopqrstuvwxyz')
```

random.shuffle(x) -- випадковим чином “перемішує” елементи послідовності (списку або рядка) x, зберігаючи результат в x.

```
mylist = [1, 0, 3, 5, 7, 1, 2]
random.shuffle(mylist)
print mylist
```

random.random() -- повертає випадкове дійсне число на проміжку від 0.0 (включно) до 1.0 (не включаючи).

```
print random.random()
```

random.uniform(a, b) -- повертає випадкове дійсне число на відрізку від a до b включно.

```
print random.uniform(0, 0.1)
print random.uniform(10, 100)
```