Python

Лекция 1

Преподаватель: *Дмитрий Косицин*
BSU FAMCS (Fall'19)

О курсе

Формат курса

- 6 лекций + 5 лабораторных + 2 домашки
- Дополнительные лекции / семинары и домашки

Что ожидается

- Все материалы доступны в anytask
- Будет много советов =)

e-mail для связи: u@trix.by

telegram для связи: @dzmitryi

О языке

...

История

Автор: Гвидо ван Россум

Язык появился в 1991 году.

- 1994 год – Python 1.0
- 2000 год – Python 2.0
- 2008 год – Python 3.0

Особенности

- Free, OpenSource, Portable (cross platform)
- Неплохая стандартная библиотека

- Высокоуровневый
- Интерпретируемый
- Объектно-ориентированный
- Жесткая динамическая типизация

Эффективность

- Понятность кода
  - Легко писать
  - Легко читать
  - Легко отлаживать
  - Есть официальный style guide (PEP-8, PEP-257)
- Низкая эффективность
  - Чистый Python медленнее C++ в 5-100 раз
- Использование
  - Прототипы
  - Интеграция / вспомогательные программы
  - Исследования
  - Web-сервисы и приложения

Python Zen

```
import this
```

- Beautiful is better than ugly.
- ...
- Simple is better than complex.
- Complex is better than complicated.
- ...
- Readability counts.
- Special cases aren't special enough to break the rules.
- ...

Версии и интерпретаторы

Версии

- Существует 2 официальные несовместимые версии Python: 2.x и 3.x
- 3.x активно развивается (последние – 3.7.4 и 3.8.0b3)
- 2.x поддерживается (2.7.16; some features are backported from Py3)
- 2.x все еще используется во множестве компаний
- Некоторые версии Python 3.x имеют несовместимые изменения

Интерпретаторы

- Cpython (официальный)
- IronPython, Jython
- PyPy, Numba, Stackless Python

Среды разработки

- SublimeText + python
- IPython [Notebook] / Jupyter
- PyCharm Community Edition
- Visual Studio with plugin (native in VS 2017)
- Visual Studio Code

Настройка и установка

Linux/MacOS

- Обычно предустановлен
- Разные версии в менеджерах пакетов
- Самая свежая ⟶ установщик/исходники с сайта

Windows

- Установщик с сайта

(можно держать несколько версий в системе)

Установка библиотек

Индекс пакетов PyPI (Python Package Index):

- Есть все популярные библиотеки
- Установка с помощью **`pip`** (out from the box для последних версий)
- Полезно заглянуть на сайты библиотек (документация / инструкции)

Возможные проблемы:

- Компиляция библиотек
- Неправильные пути в **`PATH`** и/или **`PYTHONPATH`**

Используйте **`virtualenv`** для поддержки различных версий пакетов.
Дистрибутивы: Python(x,y), Anaconda, Canopy

Основы языка

...

Интерпретатор

Обычный режим

Интерпретатор + исходный код (новый процесс с программой)

```
$ python main.py
```

Интерактивный режим

Интерпретатор без кода (исполнение кода online)

```
$ python
```

`help(...)` – справка по указанному объекту (выход: **`q`**)

`exit()` or **`quit()`** – выход

Python как калькулятор

Типы: **int (long)**, **float, complex**; *дополнительно* ***Fraction*** *и* ***Decimal***

Арифметические операции:

+ – * ** / % *(divmod)*

```
>>> -2 * 7 // 3 + 4 ** (4 % 2)
-4
```

Особенности деления:

```
>>> print(7 / 3, 7 / 3.0, 7 // 3.0)
2.3333333333 2.3333333333 2.0
```

Замечания о числах и операциях

Битовые операторы:

<< & | ^ >>

Для задания чисел в других системах счисления используются префиксы:

```
>>> 0b1001 == (1 << 3) + 1
True
```

Тип **bool** наследуется от **int** и допускает только 2 значения:

`True` (1) и **`False`** (0)

Приоритет операторов не отличается от других языков (Py2, Py3).

Отличия Python 3

Особенности деления и **print** (эти изменения из Python 3.x можно подключить в Python 2.x):

```
>>> print 7 / 3, 7 / 3.0, 7 // 3.0
2 2.33333333333 2.0
```

В Python 2.x для работы с целыми числами используется два типа – **int** и **long**, а в Python 3 – только один (**int**).

Эту особенность следует учитывать при использовании **xrange**, а также при проверке значения на целочисленность.

Отличия Python 3

Оператор матричного умножения @ (PEP465, Python 3.5+)

```
>>> x = numpy.array([ 1., 1., 1.])
>>> m = numpy.array([[ 1., 0., 0.],
                     [ 0., 1., 0.],
                     [ 0., 0., 1.]])

>>> x @ m
numpy.array([ 1., 1., 1.])
```

Логические выражения

```
>>> 0 != 0
False
>>> 2 * 2 == 4
True
>>> False or 0 > -1 and True   # False or (0 > -1 and True)
True
>>> not 0 < 1 <= 5
False
```

Переменные

```
>>> x = 1
>>> y = 2.5
>>> x + y
3.5
>>> True and y
2.5
>>> y /= 2 * x   # y = 1.25 (yep, it's a comment)
```

Условный оператор if

```
>>> if a < 0: # first way
>>>     b = -a
>>> elif a > 0:
>>>     b = a
>>> else:
>>>     b = 0

>>> b = a if a >= 0 else -a   # second way

>>> b = abs(a)   # third way
```

Цикл while

```
>>> attempt_count = 0
>>> max_attempt_count = 5
>>> while attempt_count < max_attempt_count:
>>>     # convert from str to int
>>>     x = int(input('enter an integer: '))
>>>     if x > 20:
>>>         break
>>>     attempt_count += 1
>>> else:  # if cycle hasn't been broken
>>>     print ('expected integer has not been received'
>>>            ' after %d attempts' % max_attempt_count)
```

Цикл for

```
>>> # no need to maintain counter
>>> for i in range(1, 5, 2):  # xrange Python 2.x
>>>     print(i, i + 1, end=' ')
1 2 3 4
```

- В Python 3.x **`range`** возвращает генератор, **`xrange`** отсутствует
- В Python 2.x **`range(5)`** vs **`xrange(5)`** – список / генератор (разберем позднее)

Напоминание. Крайне нежелательно изменять ***итерируемый объект*** (здесь – **`range`**) в теле цикла!

Особенности синтаксиса

В Python есть **break** и **continue**, а также ключевое слово **pass**

```
>>> while True:   # infinite loop
>>>     pass
```

Ключевое слово **else** применимо в связке с **if**, так и с **for**, **while** и **try**

Конструкция **switch** заменяется на **if-elif-…-else**

Напоминание. Наличие **else** крайне рекомендуется. В нем следует либо разместить **pass**, либо бросить исключение.

Функции

```
>>> def print_hello():
>>>     print('hello')

>>> print_hello()
hello

>>> def get_greetings(name='Bob'):
>>>     return 'Hello, ' + name

>>> print(get_greetings('Alex'))
Hello, Alex
```

Немного о типах

Пустая функция или функция с оператором **return** без аргумента возвращает специальный объект **None**

```
>>> def empty():
>>>     return
```

Можно вернуть также сразу несколько (кортеж) значений

```
>>> def multiple():
>>>     return 1, 2
```

Узнать тип объекта можно вызвав функцию **type**:

```
>>> type('hello')
<type 'str'>
```

Сравнение объектов

Все переменные – указатели на некоторые ячейки памяти.

id(…) – возвращает адрес конкретного объекта

Проверка, что **x** и **y** указывают на ***один и тот*** же объект:

x is y

Проверка, **x** и **y** указывают на ***равные по значению*** объекты:

x == y

Пример сравнения объектов

```
>>> x = [3, 5, 9]
>>> y = [3, 5, 9]

>>> print("are objects equal: %s" % (x == y))
are objects equal: True

>>> print("are objects same: %s (id of x - %d; id of y - %d)"
          % (x is y, id(x), id(y)))
are objects same: False (id of x - <some number>; id of y - <some other number>)

>>> print("x equals itself comparing values (%s)"
          " and identifiers (%s)" % (x == x, x is x))
x equals itself comparing values (True) and identifiers (True)
```

Встроенные функции

Посмотреть доступные переменные можно вызвав функцию **dir**

```
# просмотр встроенные объектов (функций, исключений и т.п.)
>>> dir()

# просмотр атрибутов и методов некоторого объекта
>>> dir(object)
```

Встроенные функции

- **bin**, **oct**, **hex** – преобразует число в строку в заданной системе счисления
- **ord**, **chr** – преобразует Unicode-символ (UCS2) в числовой код и обратно
- **bool**, **int**, **str** – стандартные типы (в т.ч. могут использоваться для преобразования типов)
- **abs**, **sum**, **round**, **min**, **max**, **pow**, **divmod** – общие математические функции

```
>>> max(1, 3, 4, 2)  # an iterable may be also passed into
4
```

Доступные математические модули: math, cmath, decimal, fractions, random (и statistics – Python 3.4+)

Списки и кортежи

Список

Список – изменяемая последовательность объектов

Создание:

```
x = list()    x = []    y = x.copy()
```

Добавление элементов (изменение списка!):

```
x.append(v)        x.insert(idx, v)
x += <iterable>    x.extend(iterable)
```

* *iterable* – некий *итерируемый* объект, например, список или кортеж

Индексирование

Взятие элемента:

```
x[idx]
```

Присваивание элемента:

```
x[idx] = v
```

Допускаются отрицательные индексы:

```
>>> x = [100]
>>> x[0] == x[len(x) - 1] == x[-1]   # в этом списке 1 элемент
True
```

Некорректные индексы порождают исключение **IndexError.**

Работа со списком

Удаление элементов (изменение списка!):

```
x.pop(idx)    x.remove(v)    x.clear()    del x[idx]
```

Изменение порядка элементов:

```
x.sort()    x.reverse()
```

Вопрос: какая сортировка используется?

Проверка принадлежности элемента:

```
x.index(v)    x.count(v)    v in x    v not in x
```

Преобразование к **bool**:

`bool([])` вернет **`False`**

Работа со списком

Списки можно сравнивать (лексикографически)

```
>>> [1, 2, 3] > [1, 2, 1]
True
```

Вопрос: какой будет результат сравнения [1, 2] и [2]?

Списки можно умножать:

```
>>> [1] * 5 == [1, 1, 1, 1, 1]
True
```

Получить отсортированную ***копию*** списка – **sorted**(x) (built-in функция, поддерживает аргумент *key* для сортировки сложных типов)

Развернуть список – **reversed**(x) (built-in, возвращает *итератор*)

Замечание: методы **copy** и **clear** появились в Python 3.x.

Кортежи

Кортеж – неизменяемая последовательность объектов

Отличие от списка: не допускает присваивания по индексу, добавления, вставки и удаления элементов, а также сортировку и обращение порядка.

Создание:

```
>>> x = (1, 2, 3)  # or just x = 1, 2, 3
>>> y = tuple([5, 6, 7])


x += 1, 2  # создает новый кортеж!
```

Словари и множества

...

Словари и хэши

Словарь отражает связь ключ-значение.

В Python словари реализованы как hash-таблица (ассоциативный массив пар).

Вопрос: какая хэш-функция используется?

Создание словаря:

```
>>> empty_dict = {}  # or empty_dict = dict()
>>> x = dict([(0, 6), (1, 7), (2, 8)])
>>> y = {2: 8, 1: 7, 0: 6}
```

Особенности хэширования

```
>>> hash(1)
1

>>> hash(True)
1

# тут все хорошо
>>> hash((1, 2, 3))
>>> hash(tuple())

# raise TypeError!
>>> hash([1, 2, 3])
>>> hash(x)
```

Важно! Ключи должны быть хэшируемыми (подробнее разберем позднее).

Работа со словарями

Взятие элемента:

```
x[key]    x.get(key, default)
```

Присваивание элемента (см. также **update**):

```
x[key] = value    x.setdefault(key, default)
```

Удаление элемента (см. также **clear**):

```
del x[key]    x.pop(key, default)
```

Проверка принадлежности:

```
key in x
```

Особенности словарей

Словари не гарантируют порядок обхода (даже в Python 3.x!)

Словари можно сравнивать на равенство.

Методы для итерирования в Python 2.x:

- **keys**(), **values**(), **items**() – возвращают списки ключей, значений и пар ключ-значение
- **iterkeys**(), **itervalues**(), **iteritems**() – возвращают, соответственно, итераторы

В Python 3.x есть только методы **keys**(), **values**() и **items**() – возвращают *view*-объекты. Такие объекты отражают изменения в исходной коллекции.

Множества

Множества (**set**) являются набором некоторых элементов: порядок не важен, все элементы хэшируемы.

Для множества есть неизменяемый аналог – **frozenset** (как для **list** – **tuple**).

```
>>> x = set()
>>> x_frozen = frozenset()
```

Неизменяемые множества (**frozenset**) можно хэшировать

```
>>> hash(frozenset())   # все хорошо
>>> hash(set())   # raise TypeError!
```

Работа с множествами

Создание:

```
set([1, 2, 3])    {1, 2, 3}
```

Добавление элемента:

```
x.add(key)
```

Удаление элемента (см. также **clear, pop**):

```
x.remove(key)    x.discard(key)
```

Проверка принадлежности:

```
key in x
```

Работа с множествами

Логические операции над множествами:

- Пересечение:

```
x & y   x.intersection(y)  |  x &= y   x.insersection_update(y)
```

- Объединение:

```
   x | y   x.union(y)   |          x |= y   x.update(y)
```

- Разность:

```
 x - y   x.difference(y)   |   x -= y   x.difference_update(y)
```

Также доступны проверка вхождения одного множества в другое (**issubset**, **issuperset**) и симметрическая разность множеств (**^**, **symmetric_difference**).

Стиль кода

...

Стиль кода

О пробелах

- Используйте ровно 4 пробела вместо tab.
- Добавляйте 2 строки между глобальными функциями, одну – между методами классов.
- Блоки **for**, **if** и пр. *желательно* визуально отделять пустой строкой
- В конце файла *желательно* добавлять пустую строку.
- Знаки математических операций следует выделять пробелами, кроме знака '=' при передаче аргументов по умолчанию
- Перед и после скобок (при вызове функции, индексировании) пробел не нужен
- После запятой пробел нужен всегда, перед – никогда

Стиль кода

Именование

var_name, function_name, ClassName, CONST_NAME

Переменные должны иметь понятные имена и доносить либо назначение переменной, либо описывать то, что в ней содержится (на что указывает).

В именах функций должен присутствовать глагол.

Длина строк

Длина строк не должна превышать 100 символов (ну или 120 – почти всегда), документации – 72

Стиль кода

Дублирующийся код следует выносить в функции.

Комментарии нужны для пояснения тонких моментов.

Кодировка файлов желательна utf-8. Можно дополнительно добавить строку в начало файла:

```
# -*- coding: utf-8 -*-
```

Замечание: все требования – выдержка из PEP-8 и Google style guide.

Утилиты для проверки

Проверка кода без выполнения:

- pep8.py
- PyChecker
- PyFlakes
- pylint
- (PyCharm)

Импорт модулей

Основы импорта модулей

Импорт модулей

Импорт модулей:

- **import** <module_name> [as <alias>]
- **from** <module_name> **import** (<name_1> [as <alias_1>],
  …,
  <name_k> [as <alias_k>])

```
>>> import sys
>>> print(sys.version)
3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:14:34) [MSC v.1900
32 bit (Intel)]
```

Загрузка модулей

Все загруженные модули хранятся в sys.modules

```
>>> def is_fractions_imported():
>>>     return 'fractions' in sys.modules

>>> assert not is_fractions_imported()

>>> import fractions
>>> assert is_fractions_imported()
```

Специальный модуль __future__ (актуален для Python 2)

```
>>> from __future__ import division
>>> print(7 / 3)
2.333333333333
```

Поиск модулей

Очередность мест поиска:

1. sys.modules (уже мог быть загружен)
2. builtins
3. sys.path:
   - current working directory
   - PYTHONPATH
   - Python source directory etc.

! Можно модифицировать sys.modules и sys.path

* на самом деле чуть сложнее

Файлы

...

Файлы

```
>>> f = open("path_to_file")  # file object
>>> # some actions
>>> f.close()
```

Если произойдет исключение до метода **close**, файл не будет закрыт.

Используйте менеджер контекстов (context manager):

```
>>> with open("path_to_file") as f:
>>>     # some actions
```

Подробнее о файлах

open(path_to_file, mode, buffering, encoding) – открывает файл для работы с ним

- **mode** – способ обращения с файлом: r, w, a, b, t, etc.
- **buffering** – размер буфера

В Python 3 файлы можно читать сразу в кодировке UFT-8 (параметр "encoding")

В Python 2 следует использовать **codecs.open**(...), **codecs.encode**(string, 'utf-8'), **codecs.decode**(string, 'utf-8')

Файлы, по сути, являются последовательностью байтов, а их интерпретация зависит от кодировки.

Работа с файлами

Чтение: методы **read**, **readline**

Запись: метод **write**

Итерирование: **for** line **in** f: ...

Также можно смещаться по файлу (**seek**) и сбрасывать буффер (**flush**).

Важно! Flush не гарантирует запись на диск. Используйте **os.fsync**.

Стандартные потоки ввода-вывода: **sys.stdout** и **sys.stderr**.

Форматы ввода-вывода

В стандартной библиотеке Python поддерживается работа:

- архивами (**zipfile**, **gzip/zlib**)
- **csv**-файлами (самый простой формат таблиц)
- **json** (удобно сохранять словари – json.dumps/json.loads)

Также есть сторонняя библиотека для работы с **yaml** – расширенный и более читаемый **json**.

Для красивого форматирования объектов при печати есть модуль **pprint**.

Строки

Строки

Python 2	Python 3
str – кортеж байтов **bytearray** – изменяемая последовательность байтов **unicode** – unicode строка	**bytes** – кортеж байтов **bytearray** изменяемая последовательность байтов **str** – unicode строка

Могут быть заданы как в одинарных '', так и двойных "" кавычках. Есть также строки в трех кавычках подряд (***docstring***).

К строкам могут быть применены литералы: **r**"", **u**"", **b**""

Длинную строку можно разместить на нескольких строках, взяв ее в скобки.

Методы работы со строками

Методы работы как с кортежами: сложение, умножение, проверка вхождения, длина, индексация:

```
>>> x = 'abc'
>>> 'b' in x
True
```

Вопрос: Какой алгоритм используется в стандартной библиотеке для поиска подстрок в строке?

```
>>> x += 'd' * 2  # x: 'abcdd'
>>> len(x)
5

>>> x[:2]
ab
```

Методы работы со строками

Поиск подстрок: **index**, **count** и **find/rfind**

```
>>> x.find('z')
-1
```

Проверка вхождения: **startswith, endswith**

```
>>> x.endswith(('aa', 'dd'))   # можно только один аргумент
True
```

Замена подстрок

```
>>> x = x.replace('a', 'z')   # 'zbcdd'; return a new one
```

Методы работы со строками

Объединение/разбиение по символу: **join**, **split/rsplit** (см. max_split аргумент)

```
# 'zbcdd' -> ['zb', 'dd'] -> 'zbcdd'
>>> 'c'.join(x.split('c'))
zbcdd
```

Важно! Если требуется объединить большое количество строк, следует использовать ''.**join**(список_строк)

Очистка строк: **strip/lstrip/rstrip**

```
>>> x.strip('d')
zbc
```

Методы работы со строками

Преобразование регистра: **lower**, **upper**, **title**, **capitalize**.

Кодирование: **encode**, **decode**.

Проверка символов: **isalpha**, **isdigit**, etc.

Замечание. Строки, содержащие все цифры или все знаки пунктуации определены в модуле string.

Замечание. Работа с **base64** организуется с помощью библиотеки base64.

Модель памяти

Изменяемые и неизменяемые объекты. Копирование. Синглтоны.

Объекты

Все сущности в Python являются объектами (наследниками типа ***object***):

- Числа, списки, словари
- Классы и функции
- Код

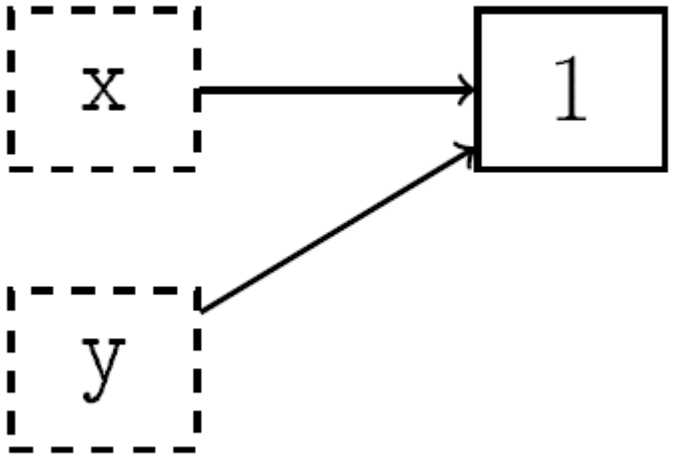
У объектов есть свойства:

- *Идентичность* – адрес в памяти (функция **id**), не изменяется, для сравнения используется **is** и **is not**
- *Тип* определяет допустимые значения и операции над ними, также не изменяется (см. функцию **type**)
- *Значение* может быть *изменяемым* (list) и *неизменяемым* (tuple)

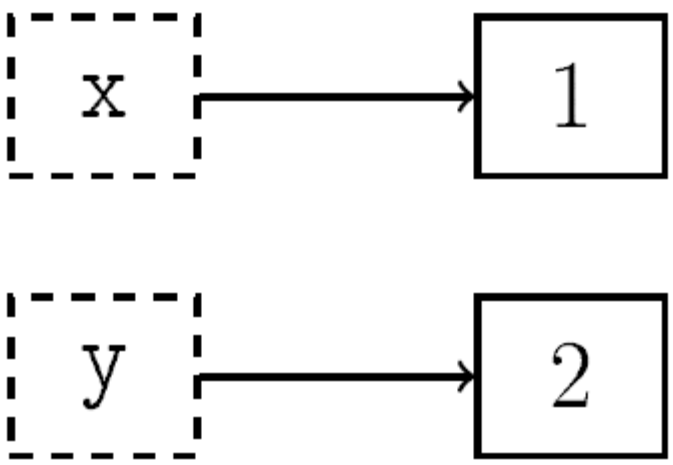
Связь имени и объекта

Каждому имени в коде соответствует объект в памяти:

```
x = 1
y = x
```



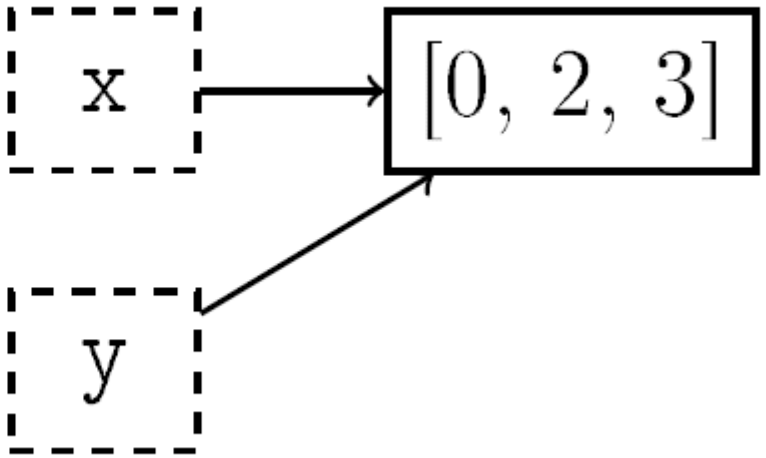

```
x = 1
y = x
y = 2
```



Изменяемые объекты

Значение объектов может меняться:

```
x = [1, 2]
y = x
y += [3]
x[0] = 0
```



Неизменяемые объекты

Значение некоторых объектов, например, кортежей – нет:

```
>>> a = (1, 2)
>>> a[0] = 0
TypeError: 'tuple' object does not support item assignment
```

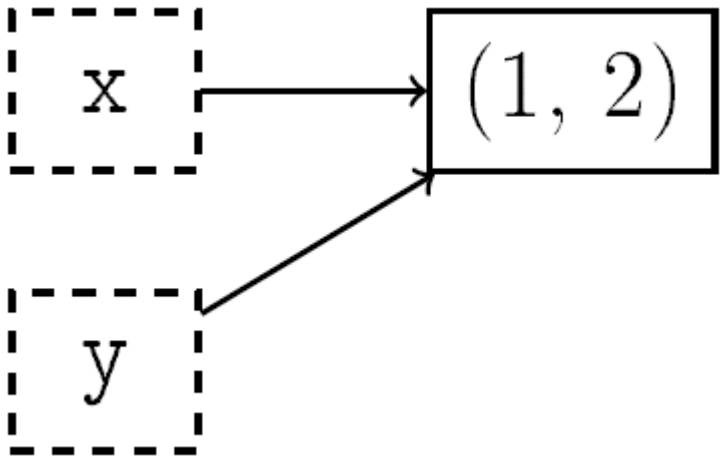
Вопрос: что будет в таком примере?

```
>>> a = ([0], )
>>> a[0] += [1]
```

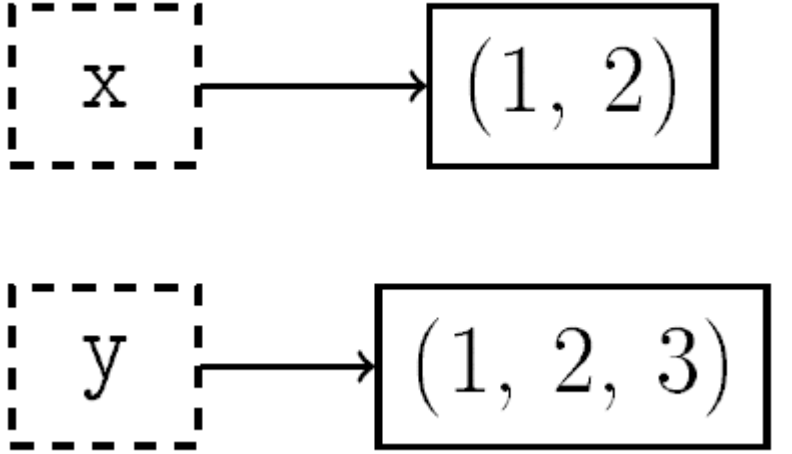
Неизменяемые объекты

При работе с неизменяемыми объектами некоторые операции создадут новый объект:

```
x = (1, 2)
y = x
```



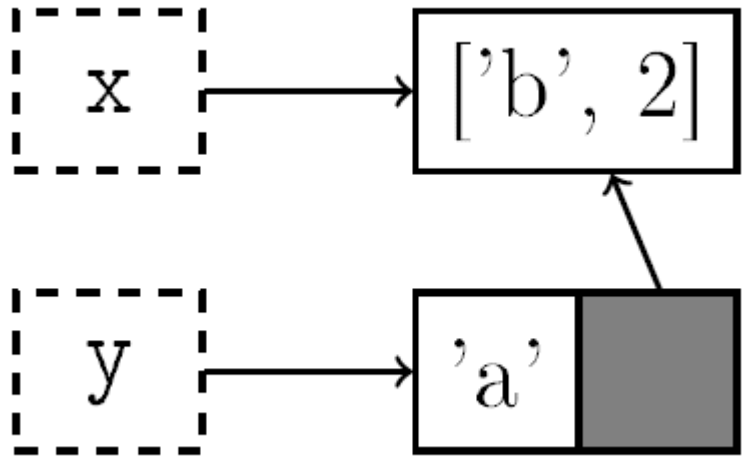

```
x = (1, 2)
y = x
y += (3,)
```



«Контейнеры»

Объекты, например, списки и словари, могут содержать ссылки на другие объекты:

```
x = [1, 2]
y = ('a', x)
x[0] = 'b'
```



Копирование объектов

Копирование объектов

- неизменяемые: при присваивании создается копия
- изменяемые: присваивается ссылка, поэтому нужно использовать либо методы **copy** и **deepcopy** (модуль copy), либо конструктор копирования

Неизменяемые типы

int float complex bool str tuple frozenset

Изменяемые типы

list dict set

Синглтоны

Некоторые объекты поддерживаются в единственном экземпляре:

None True False

А также

- **NotImplemented** (используется в операторах сравнения)
- **Ellipsis** (используется в математических библиотеках)

Замечание. В этом можно убедиться вызвав функцию **id**.

Помимо **None**, **True**, **False**, **NotImplemented** и **Ellipsis**, числа от **-5** до **256** могут быть также синглтонами ввиду их частого использования. Это зависит от *реализации интерпретатора*.

Синглтоны

Важно! Объект **None** используется в качестве параметра по умолчанию для обозначения того, что ничего не передано.

Если значение **None** является допустимым и его нельзя использовать в качестве аргумента по умолчанию, то создают объект (например, глобальный) типа **object** и проверяют, является ли переданный аргумент идентичным этому объекту.

Замечание. При каждой загрузке модуля объект создаваться заново не будет.

```
>>> not_set = object()
>>> def f(x=not_set):
>>>     is_x_set = (x is not not_set)
```

Особенности работы с коллекциями

Упаковка и распаковка переменных.
Особенности итерирования по нескольким коллекциям. Слайсы.
Функциональная обработка коллекций.

Упаковка переменных

Упаковка – создание кортежа / списка / т.п.

```
>>> x = [1, 2]
>>> x0 = x[0]
>>> x1 = x[1]
>>> print(x[0], x[1])
1, 2
```

Пример распаковки кортежа (пары значений):

```
>>> for x, y in zip(range(5), range(5, 10)):
>>>     print(x, y)
```

Распаковка переменных

Кортеж можно распаковать автоматически:

```
>>> x0, x1 = x
>>> print(x0, x1)
1, 2
```

Для обмена переменных местами – упаковать и распаковать в другом порядке:

```
>>> x[0], x[1] = (x[1], x[0])   # скобки не обязательны
>>> print(x[0], x[1])
2, 1
```

Распаковка переменных

Важно! Присваивание выполняется справа налево, но подвыражения в присваивании – в порядке следования:

```
>>> x = [1, 2]
>>> i = 0
>>> i, x[i] = 1, 1
>>> print(x)
[1, 1]
```

Распаковывать можно **list**, **tuple**, **set**, а также итераторы.

Распаковка переменных

Допускается вложенность:

```
>>> ((x, _), z) = [[1, 2], 3]
>>> print(x, z)
1, 3
```

Важно! Символ ‘,’ (запятая) является элементом синтаксиса не только кортежей – например, при бросании исключений распаковывать кортеж неявно нельзя.

В Python 3 допустима частичная распаковка (PEP-3132):

```
>>> head, *middle, tail = range(5)
>>> print head, middle, tail
0, [1, 2, 3], 4
```

Итерирование по нескольким коллекциям

Итерирование по двум коллекциям – функция **zip**:

```
>>> for x, y in zip([1, 2, 3], [-3, -2, -1]):
>>>     print(x % y == 0, end=' ')
False True True
```

Важно! zip возвращает новый **список кортежей** в Python 2.x и **генератор** в Python 3.x.

Вопрос: что будет, если коллекции разной длины?

enumerate(x) – сокращение **zip**(**range**(**len**(x)), x)

Слайсы

Можно обращаться сразу к нескольким элементам коллекции:

```
>>> x = list(range(10))
>>> print(x[0:10:2])
0 2 4 6 8
```

Замечание. Вспомните range(0, 10, 2)

Слайсы за пределами коллекции или некорректные:

```
>>> x[100:110]
[]  # тип соответствует типу переменной x

>>> x[0:10:-2]
[]
```

Способы задания слайсов

Следующие записи эквивалентны:

- `x[9:0:-2]`
- `x[-1:0:-2]`
- `x[:None:-2]`

Слайс – это объект **slice**

```
>>> x[slice(9, 0, -2)] == x[9:0:-2]
True
```

Замечания по работе со слайсами

Слайсу можно дать имя и связать с переменной

```
>>> even_indices_slice = slice(None, None, 2)
>>> print(x[even_indices_slice])
```

Слайсам можно присваивать, в том числе iterable с иным количеством элементов:

```
>>> x = [1, 2, 3]
>>> x[0:2] = [7, 6, 5]
>>> print(x)
[7, 6, 5, 3]
```

Замечания по работе со словарями

Особенности создания словарей с ключами, имеющими одинаковый хэш:

```
>>> x = {1: 'a', True: 'b', 1.0: 'c'}
>>> assert x == {1.0: 'c'}
```

Методы для итерирования в Python 2.x:

- **keys**(), **values**(), **items**() – возвращают списки ключей, значений и пар ключ-значение
- **iterkeys**(), **itervalues**(), **iteritems**() – возвращают, соответственно, итераторы
- **viewkeys**(), **viewvalues**(), **viewitems**() – возвращают view-объекты

В Python 3.x методы **keys**(), **values**() и **items**(), возвращают на самом деле *view*-объекты. Такие объекты отражают изменения в исходной коллекции.

Замечания по работе со списками

Три способа создать список, содержащий три списка:

```
>>> x = [[], [], []]
>>> y = [[]] * 3
>>> z = []
>>> for _ in range(3):
>>>     z.append([])
([[], [], []], [[], [], []], [[], [], []])

# использовать ";" (semicolon) нельзя!
>>> x[0].append(1); y[0].append(2); z[0].append(3)
```

Замечания по работе со списками

```
>>> print(x, y, z)
([[1], [], []], [[2], [2], [2]], [[3], [], []])
```

Wow!.. *Лучше использовать другой способ!*

```
>>> y_pretty = [[] for _ in range(3)]
>>> y_pretty[0].append(2)
>>> print(y_pretty)
[[2], [], []]
```

Такая конструкция называется list-comprehension.

Функциональный подход

Основные функции для работы с последовательностями:

- **map** – применить функцию к каждому элементу последовательности;
- **filter** – оставить только те элементы, для которых переданная функция возвращает **True** (предикатом может быть **None**);
- **all** – возвращает **True**, если все элементы преобразуются к **True**;
- **any** – возвращает **True**, если хотя бы один элемент **True.**

Важно! В Python 2.x функции **map** и **filter** возвращают списки, в то время как в Python 3.x – генераторы.

Comprehensions

Comprehensions допускают вложенные **for** и одно **if** выражение:

```
>>> def is_odd(x):
>>>     return bool(x % 2)

>>> s1 = filter(is_odd, range(10))
>>> s2 = [x for x in range(10) if is_odd(x)]
>>> assert list(s1) == s2
```

Tuple-comprehension нету. Выражение с круглыми скобками – генератор

```
>>> s3 = list(x for x in range(10) if is_odd(x))
>>> assert s1 == s2 == s3
```

Comprehensions

Есть comprehension-выражения для создания словарей и множеств:

```
>>> d = {x: y**2
...      for x in range(2)
...      for y in range(x + 1)}
>>> assert d == {0: 0, 1: 1}


>>> s = {1 for _ in range(10)}
>>> assert s == set([1])
```

Замечание. Перед **for** может стоять *любое* допустимое выражение, в том числе содержащее некоторое преобразование (см. `y**2`) *или* условие.

Замечание. Comprehension-выражения допускают произвольное количество **for** и **if**, причем ***не*** обязательно поочередно: после **for** допустимо несколько **if**'ов.

Statement **del**

Statement **del** имеет несколько смысловых нагрузок:

- «разрывает связь» между переменной и объектом (здесь также задействуется счетчик ссылок объекта)
- удаляет элемент, атрибут или слайс (за удаление отвечает сам объект)

```
>>> class X(object):
>>>     a = None
>>>
>>> del X.a
>>>
>>> y, z = [1, 2, 3], {'x': 0}

>>> del y[0], z['x']
```

Вопрос: как с помощью **del** очистить список, не удалив при этом объект?

Ответы и полезные ссылки

Ответы

- Сортировка – TimSort (производная merge sort и insertion sort)
- Hash-функция в Python < 3.4 – FNV, далее – SipHash (PEP456)
- В стандартной библиотеке для поиска подстрок в строке используется алгоритм Бойера-Мура

Ответы

В случае передачи в метод **zip** коллекций (итераторов по коллекциям) разной длины, итерирование закончится при достижении конца наименьшей коллекции.

В случае итераторов важно, какой итератор будет исчерпан первым.

```
>>> i1, i2 = iter(range(5)), iter(range(2))

>>> for x, y in zip(i1, i2):
...     pass

>>> assert next(i1) == 3
```

Ответы

Очистить список, не удалив его самого, можно так:

```
>>> x = []
>>> del x[:]
>>> x[:] = []   # эквивалентная запись
```

Полезные ссылки

- Сайт Jupyter (инструкция по установке и документация – https://jupyter.org
- Как использовать сразу две версии Python: здесь и здесь
- Документация по «магическим» выражениям Jupyter – здесь

Финальные замечания

- Нет скобок – все регулируется отступами (4 пробела, см. policy)
- Оператор **for** для итерирования – аналог ***foreach***
- Стандартная библиотека содержит много полезного:

  https://docs.python.org/2.7/library | https://docs.python.org/3/library