Python

Лекция 4

Преподаватель: *Дмитрий Косицин*
BSU FAMCS (Fall'18)

Итераторы и генераторы

Последовательности. Итерируемые объекты. Итераторы. Генераторы. Дополнительные способы итерирования.

Менеджеры контекста

Модули стандартной библиотеки

Работа с файловой системой. Регулярные выражения. Модуль functools.

Математические библиотеки и работа с данными

Numpy. SciPy. Sympy. Matplotlib и Seaborn. Pandas.

Работа с Web. GUI

Urllib, Requests. lxml, BeautifulSoup. SQLAlchemy. Django, Flask. Tkinter. PyQT. Kivy.

Итераторы и генераторы

Последовательности. Итерируемые объекты. Итераторы. Генераторы. Дополнительные способы итерирования.

Sequence and iterable

Последовательность (*sequence*) – упорядоченный *индексируемый* набор объектов, например, **list**, **tuple** и **str**.

У этих объектов переопределены «магические методы» ***__len__*** (длина последовательности) и ***__getitem__*** (отвечает за индексацию).

Итерируемое (*iterable*) – упорядоченный набор объектов, элементы которого можно получать по одному.

У таких объектов реализован метод ***__iter__*** – возвращает итератор, который позволяет обойти итерируемый объект.

Итераторы

Итератор (iterator) представляет собой «поток данных» – он позволяет обойти все элементы *итерируемого* объекта, возвращая их в некоторой последовательности.

В итераторе переопределен метод ***__next__*** (***next*** в Python 2), вызов которого либо возвращает следующий объект, либо бросает исключение ***StopIteration***, если все объекты закончились.

Для явного получения итератора и взятия следующего элемента используются *built-in* методы **iter** и **next**.

```
for item in sequence:
    action(item)
```

```
def for_sequence(sequence, action):  # "for" for sequence
    i, length = 0, len(sequence)
    while i < length:
        item = sequence[i]
        action(item)
        i += 1
```

```
def for_iterable(iterable, action):  # "for" for iterator
    iterator = iter(iterable)
    try:
        while True:
            item = next(iterator)
            action(item)
    except StopIteration:
        pass
```

Итераторы

Итераторы представляют собой классы, содержащие информацию о текущем состоянии итерирования по объекту (например, индекс).

После обхода всех элементов итератор «истощается» (exhausted), бросая исключение ***StopIteration*** при каждом следующем вызове ***__next__***.

Замечание. Функция ***next*** имеет второй параметр – значение по умолчанию, которое будет возвращено, когда итератор исчерпается.

Замечание. У функции ***iter*** также есть второй аргумент – значение, до *получения* которого будет продолжаться итерирование.

Пример реализации итератора

```
class RangeIterator(collections.Iterator):
    def __init__(self, start, stop=None, step=1):
        self._start = start if stop is not None else 0
        self._stop = stop if stop is not None else start
        self._step = step  # positive only

        self._current = self._start

    def __next__(self):
        if self._current >= self._stop:
            raise StopIteration()

        result = self._current
        self._current += self._step
        return result
```

Пример использования итератора

Поскольку итераторы хранят информацию о состоянии, их можно прервать и впоследствии продолжить итерироваться. *Вопрос: что выведет следующий код?*

```
>>> odd_indices_iterator = RangeIterator(1, 10, 2)
>>>
>>> for idx in odd_indices_iterator:
>>>     if idx > 5:
>>>         break
>>>     print(idx)
>>>
>>> for idx in odd_indices_iterator:
>>>     print(idx)
```

Итерируемые и истощаемые

Последовательности *итерируемы* и *не истощаемы* (можно много раз итерироваться по ним).

Итерируемые объекты (не последовательности) могут как не истощаться (**range** в Py3/**xrange** в Py2), так и истощаться (генераторы).

Итераторы *итерируемы* (возвращают сами себя) и *истощаемы* (можно только один раз обойти).

Замечание. Зачастую в классах не реализуют отдельный класс-итератор. В таком случае метод ***__iter__*** возвращает *генератор*.

Замечание. В Python 2 **range** возвращает список, а **xrange** – генератор.

Пример итерируемого объекта

```
class SomeSequence(collections.Iterable):
    def __init__(self, *items):
        self._items = items

    def __iter__(self):
        for item in self._items:
            yield item

    def __iter__(self):
        yield from self._items  # только в Python 3.3+

    def __iter__(self):  # простой и менее гибкий вариант
        return iter(self._items)
```

Напоминание. В модуле **collections** есть и другие базовые классы, например **Sequence**. Эти классы реализуют множество полезных методов, требуя переопределить лишь несколько.

Генератор

Генератор – итератор, с которым можно взаимодействовать (PEP-255).

Каждый следующий объект возвращается с помощью выражения **yield**. Это выражение *приостанавливает* работу генератора и передает значение в вызывающую функцию. При повторном вызове исполнение продолжается с *текущей* позиции либо до следующего **yield**, либо до конца функции.

Генераторы удобно использовать, когда вся последовательность сразу не нужна, а нужно лишь по ней итерироваться.

```
>>> assert all(x % 2 for x in range(1, 10, 2))
```

Замечание. Выражения-генераторы имеют вид comprehensions с круглыми скобками. При передаче в функцию дополнительные круглые скобки не нужны.

Замечания по генераторам

Конструкция **yield from** делегирует, по сути, исполнение некоторому другому итератору (Python 3.3+, PEP-380).

В Python 3 появилась возможность у генераторов (например, **range**) узнать длину генерируемой ими последовательности (метод ***__len__***) и проверить, генерируют ли они определенный элемент (метод ***__contains__***).

В Python 2 ввиду реализации **xrange** не принимает числа типа **long**.

Также в Python 3 есть специальный класс – **collections.ChainMap**, который представляет собой обертку над несколькими mapping'ами.

Дополнительные способы итерирования

В стандартной библиотеке есть модуль **itertools**, в котором реализовано множество итераторов:

- **cycle** – зацикливает некоторый iterable
- **count** – бесконечный счетчик с заданным начальным значением и шагом
- **repeat** – возвращает некоторое значение заданное число раз

Также есть комбинаторные итераторы:

- **product** – итератор по декартову произведению последовательностей (по сути, генерирует кортежи, если бы был реализован вложенный ***for***)
- **combinations** – итератор по упорядоченным сочетаниям элементов
- **permutations** – итератор по перестановкам переданных элементов

Дополнительные способы итерирования

- **chain** – итерируется последовательно по нескольким iterable
- **zip_longest** – аналог zip, только прекращает итерироваться, когда исчерпывается не первый, а последний итератор
- **takewhile/dropwhile/filterfalse/compress** – отбирает элементы последовательности в соответствии с предикатом
- **islice** – итераторный аналог slice (не создает списка элементов)
- **groupby** – группирует последовательные элементы
- **starmap** – аналог **map**, только распаковывает аргумент при передаче
- **tee** – создает ***n*** копий итератора

Замечание. В Python 2 доступны **ifilter** и **izip** – итераторные аналоги **filter** и **zip**.

Замечание. В Python 3.2 появилась функция **accumulate**, которая возвращает итератор по кумулятивному массиву.

Менеджеры контекста

...

Менеджеры контекста

В процессе работы с файлами важно корректно работать с исключениями: файл необходимо закрыть в любом случае.

Данный синтаксис позволяет закрыть файл по выходе из блока **with**:

```
with open(file_name) as f:
    # some actions
```

Функция **open** возвращает специальный объект – *context manager*. Менеджер контекста последовательно *инициализирует* контекст, *входит* в него и корректно обрабатывает *выход*.

Пример менеджера контекста

```
class ContextManager(object):
    def __init__(self):
        print('__init__()')

    def __enter__(self):
        print('__enter__()')
        return 'some data'

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('__exit__({}, {})'.format(
            exc_type.__name__, exc_val))


with ContextManager() as c:
    print('inside context "%s"' % c)
```

Менеджер контекста

Менеджер контекста работает следующим образом:

- создается и инициализируется (метод **__init__**)
- организуется вход в контекст (метод **__enter__**) и возвращается объект контекста (в примере с файлом – объект типа **file**)
- выполняются действия внутри контекста (внутри блока **with**)
- организуется выход из контекста с возможной обработкой исключений (метод **__exit__**)

В примере будет выведено следующее:

```
__init__()
__enter__()
inside context "some data"
__exit__(None, None)
```

Менеджер контекста

Замечание. Если исключения не произошло, то параметры, передаваемые в функцию **__exit__** – тип, значение исключения и ***traceback*** – имеют значения **None**.

Замечание. Менеджер контекста, реализуемый функцией **open**, по выходе из контекста просто вызывает метод ***close*** (см. декоратор *contextlib.closing*).

Менеджеры контекста используются:

- для корректной, более простой и переносимой обработки исключений в некотором блоке кода
- Для управления ресурсами

Декоратор *contextlib.contextmanager* позволяет создать менеджер контекста из функции-генератора, что значительно упрощает синтаксис.

Менеджер контекста из генератора

```
>>> @contextlib.contextmanager
>>> def get_context():
>>>     print('__enter__()’)
>>>     try:
>>>         yield 'some data’
>>>     finally:
>>>         print('__exit__()’)
>>>
>>> with get_context() as c:
>>>     print('inside context "%s"' % c)
__enter__()
inside context "some data"
__exit__()
```

Модули стандартной библиотеки

Работа с файловой системой. Регулярные выражения. Модуль functools.

Работа с файловой системой

В стандартной библиотеке есть несколько модулей, отвечающих за файловую систему.

Модуль **os.path** служит, в основном, для работы с путями:

join | **abspath** | **relpath** | **commonprefix** | **split** | **normpath**

walk | **getsize** | **exists** | **isfile**

Для работы с файловой системой используется модуль **os**:

listdir | **mkdir** | **makedirs** | **remove** | **rmdir** | **rename** | **stat**

Работа с файловой системой

Для копирования и удаления файлов используется модуль **shutil**:

copy | move | rmtree

Модули **glob** и **fnmatch** предназначены для поиска файлов и папок по шаблонам с wildcard'ами, для сравнения файлов есть модуль **filecmp**.

В Python 3.6 появились полноценная библиотека для работы одновременно с путями и файловыми объектами – Pathlib (PEP-428), а также функция os.scandir – улучшенный аналог os.walk (PEP-519).

Работа с системой

Модуль **os** также содержит множество констант и функций для работы с системой:

chdir | getcwd | getenv | putenv | unsetenv
environ | extsep

Все функции являются *system specific* и могут отсутствовать. Для управления процессами есть **abort** и **kill**.

В Python 3 был систематизирован весь протокол работы с файлами / стримами – см. модуль **io**.

Полезным в модуле **io** может быть файловый буффер **StringIO** (в Py2 отдельно).

Модули re и functools

В модуле **re** собраны функции для работы с регулярными выражениями:

- **search** / **match** / **finditer** – поиск шаблона, возвращают **MatchObject**
- **split** / **sub** – разбиение строки по шаблону / замена подстроки

MatchObject имеет следующие свойства: **groups**, **groupdict**, **start**, **end**, **span** и **pos**.

В модуле **functools** содержатся полезные функции:

- ***partial*** – возвращает новую функцию, фиксируя некоторые аргументы (замена, например **lambda** x: f(x, True); также ***partialmethod*** в Py3.4+)
- ***lru_cache*** – декоратор для кэширования результатов функции в LRU-кэше (Py3.2+)
- ***singledispatch*** – декоратор, позволяющий вызывать функцию в зависимости от типа ее аргумента (Py3.4+, примеры)

Математические библиотеки и работа с данными

Numpy. SciPy. Sympy. Matplotlib и Seaborn. Pandas.

Numpy

В библиотеке Numpy реализован класс **ndarray** – представление многомерного массива.

Он характеризуется данными (data) и информацией о данных:

- Тип данных и его размер
- Смещение данных в буфере
- Размерности (shape) и размер в байтах
- Количество элементов для перехода к следующему элементу в измерении (по оси – stride)
- Порядок байтов в массиве
- Флаги буфера данных
- Ориентация данных (C-order или Fortran-order)

NDArray

NDArray соответсвует буферу – С-массиву, выровненному по размеру элемента (**itemsize**).

К отдельным элементам массива можно обращаться с помощью методов **item/itemset** – это быстрее, чем по индексу через **__getitem__**.

Очень важно! Все операции с массивом могут быть как с копированием данных, так и без. Некоторые операции возвращают *views* – новые массивы, которые указывают ***на те же данные***, но содержат о них иную информацию. При изменении данных во *view*, данные в оригинальном массиве меняются.

Размерность массива

Форма массива задается атрибутом **shape** – кортеж размерностей.

Изменить размер можно:

- ndarray.reshape() – *view*, но shape обязан быть *compatible* с текущим
- ndarray.resize() – *inplace*, может понадобиться копирование
- ndarray.shape – *inplace*, исключение, если не *compatible*

Разворачивание массива в 1-D:

- ndarray.ravel – *view*
- ndarray.flatten – *copy*
- ndarray.flat – *итератор* по flattened массиву

Замечание. Допустим 0-D массив.

Оси

Оси – составляющие общей размерности массива.

Важно! Нумерация осей (axes) ведется с нуля и соответствует декартовым координатам. Значение **None** в функциях соответствует развернутому массиву.

Изменение осей не приводит к копированию данных.

Методы **transpose** и **swapaxis** позволяют изменить порядок следования осей.

Важно! Эти методы могут сделать отображение данных не непрерывным.

Индексация

Numpy поддерживает два вида индексации: простую и «продвинутую».

Простая индексация – один элемент или слайс:

```
>>> x = numpy.arange(10)
>>> x[1], x[-2], x[3: 7]
```

В многомерном массиве элементы задаются через запятую:

```
>>> x = numpy.arange(100).reshape(5, 5, 4)
>>> x[1, 2, 3], x[1, 1:, -1], x[…, 2] == x[:, :, 2]
```

Замечание. Многоточие '…' – **Ellipsis** позволяет пропустить некоторые измерения, предполагая, что их нужно взять целиком.

Замечание. В numpy применяется index broadcasting: если массив по одной из осей имеет длину 1, он расширяется до необходимой длины (например, можно сложить с числом).

Продвинутая индексация

Очень важно! Запись x[ind_1, …, ind_n] эквивалентна x[(ind_1, …, ind_n)], но кардинально отличается от x[[ind_1, …, ind_n]].

Если в метод **__getitem__** передан ***список*** или ***ndarray***, то используется «продвинутая» индексация: она создает копию массива с указанными элементами.

```
>>> x = numpy.arange(4)
>>> x[[1, 3]] == x[[False, True, False, True]]
```

Список, переданный в качестве индекса, может содержать как индексы, так и массив **bool**, означающий, какие элементы нужно взять.

Замечание. Массив **bool** может иметь длину меньше, чем длина исходного массива. Если длина больше и есть **True** за пределами массива, будет исключение.

Универсальные функции

В numpy реализовано множество функций по работе с данными – сложение, умножение, вычисление синуса и т.п. Все операции над массивами выполняются **поэлементно**!

```
>>> x, y = numpy.arange(3), numpy.linspace(2, 3, num=3, endpoint=True)
>>> x + y == numpy.array([2, 3.5, 5])
>>> numpy.max(x) == x.max()  # 2
```

Универсальные функции (ufunc) выполняются с учетом расположения данных. Во время прохода также используется буферизация! Поэтому они работают быстрее Python built-in. На небольших данных ufunc работают медленнее, ввиду необходимости настройки.

Типы и записи

В numpy используется своя система типов, в частности, numpy.bool отличается от Python built-in bool.

В массивах помимо скалярных типов можно хранить произвольные типы dtype – по сути, C-структуры, содержащие некоторые скалярные типы.

```
>>> dt = np.dtype([('name', np.str_, 16),
                   ('grades', np.float64, (2,))])
>>> dt = np.dtype({'names': ['r','g','b','a'],
                   'formats': [uint8, uint8, uint8, uint8]})
```

В dtypes указаны имена – массивы с такими типами будут являться записями (numpy.recarray), так что к столбцам можно будет обращаться по имени.

Возможности Numpy

В Numpy есть поддержка массивов с пропусками – **MaskedArray**. Значения в таким массивах могут иметь специальное значение **numpy.ma.masked**.

```
>>> x = numpy.array([1, 2, 3, -1, 5])
>>> mx = numpy.ma.masked_array(x, mask=[0, 0, 0, 1, 0])
>>> mx.mean()  # 2.75
```

Такие значения обрабатываются функциями и не влияют на результат.

Работа с произвольными Python-функциями может быть организована:

- **apply_along_axis** (**apply_over_axes**) – применяет функцию вдоль оси (осей)
- **vectorize** – обобщенный класс функций (можно использовать как декоратор, реализован как **for**)
- **frompyfunc** – позволяет создать ufunc из обычной функции

Возможности Numpy

Также в numpy реализована работа

- с матрицами (Matrix)
- со случайными величинами (random)
- со статистическими функциями
- допустима стыковка массивов

SciPy

В тот время как Numpy реализует определенные типы данных и базовые операции, SciPy реализует множество вспомогательных функций:

- Clustering – реализация kmeans и пр.
- Constants – множество констант, математических и физических
- FFTPack – функции, выполняющие дискретное преобразование Фурье
- Integrate – интегрирование (квадратурные формулы, с фиксированной сеткой) и дифференцирование (Рунге-Кутта и пр.)
- IO – работа с форматами данных, в т.ч. MatLab
- Linalg – линейная алгебра (решение СЛАУ, разложения матриц, нахождение собственных значений, матричные функции, специальные виды матриц и пр.)
- NDImage – функции по работе с изображениями (свертки, аффинные преобразования и т.п.)

SciPy

- ODR – orthogonal distance regression
- Optimize – оптимизация (методы первого и второго порядка, метод Ньютона, BGFS, Conjugate Gradient и пр., поиск корней, МНК)
- Signal – методы обработки сигналов (звука – свертки, сплайны, фильтры)
- Sparse и Sparse.Linalg – разреженные матрицы и методы по работе с ними
- CSGraph – методы по работе со сжатыми разреженными графами (в т.ч. алгоритмы Дейкстры, BFS, DFS и т.п.)
- Spatial – работа с точками на плоскости (KDTree и пр.)
- Stats и Stats.Mstats – работа с распределениями и статистиками

Matplotlib, Seaborn, SymPy, Pandas

Для отображения данных используется библиотеки **MatplotLib** и **Seaborn**. Работать с ней так:

- Открыть MPL tutorials (или reference) или SBS tutorials (или API)
- Найти нужный пример
- Модифицировать его под свои нужды

Библиотека **Sympy** позволяет производить символьные вычисления, а **Pandas** – работать с данными (более удобная обертка над **NumPy**, ориентированная на группировку / преобразование данных и статистики).

Полезно: https://github.com/jrjohansson/scientific-python-lectures

Работа с Web. GUI

Urllib, Requests. lxml, BeautifulSoup. SQLAlchemy. Django, Flask.
Tkinter. PyQT. Kivy.

Web-запросы и парсинг

В стандартной библиотеке есть модули для работы с web:

- **socket** – работа с *сокетами*
- html.parsers; **xml.etree**, xml.dom и xml.sax – html и xml *парсеры*
- **urllib** – http-*запросы*

В то же время есть множество библиотек (зачастую, надстроек либо над стандартными модулями, либо над другими библиотеками), которые использовать гораздо удобнее.

Для парсинга html / xml популярны библиотеки **lxml** и **BeautifulSoup**.

Web-запросы и парсинг

Для выполнения *запросов* (get / post), используют **requests** – использование в ней классов и менеджеров контекста упрощает работу. Библиотеку retry используют для повторения запросов (специфика соединений в web).

Замечание. При работе с запросами не забывайте про *timeout* и *retry*!

Также существует множество web-фреймворков (для написания сайтов). Полный список здесь, а вот дни из самых популярных:

- Django (и Django REST Framework для работы с REST API);
- Falcon (очень производительный);
- Flask

Для генерации страниц (шаблонов) используют Jinja2 и genshi.

Web-запросы и парсинг

Для работы с базами данных используют **SQLAlchemy**. Для работы, например, с PostgreSQL есть своя библиотека – Psycopg. В стандартной библиотеке также есть **sqlite3** для работы с SQLite databases.

Для работы с URL можно использовать библиотеку furl.

Для выполнения множества асинхронных запросов – gevent.

Для отслеживания падений сервиса используют sentry.

Замечание. Некоторые примеры см. в ipynb-приложении.

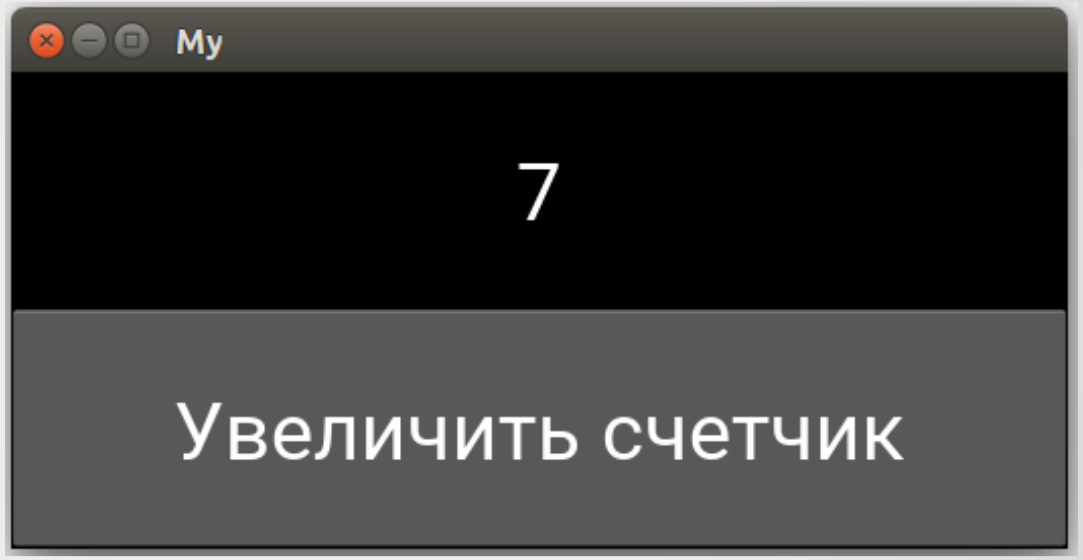
GUI

Для построения GUI на Python зачастую используют следующие библиотеки:

- **Tkinter** (стандартная)
- **PyQt/PySide**
- **wxPython**
- **PyGTK**
- **pygame** (в том числе для игр)
- **Kivy** (в том числе для мобильных платформ; статья)

Полный список здесь и здесь.

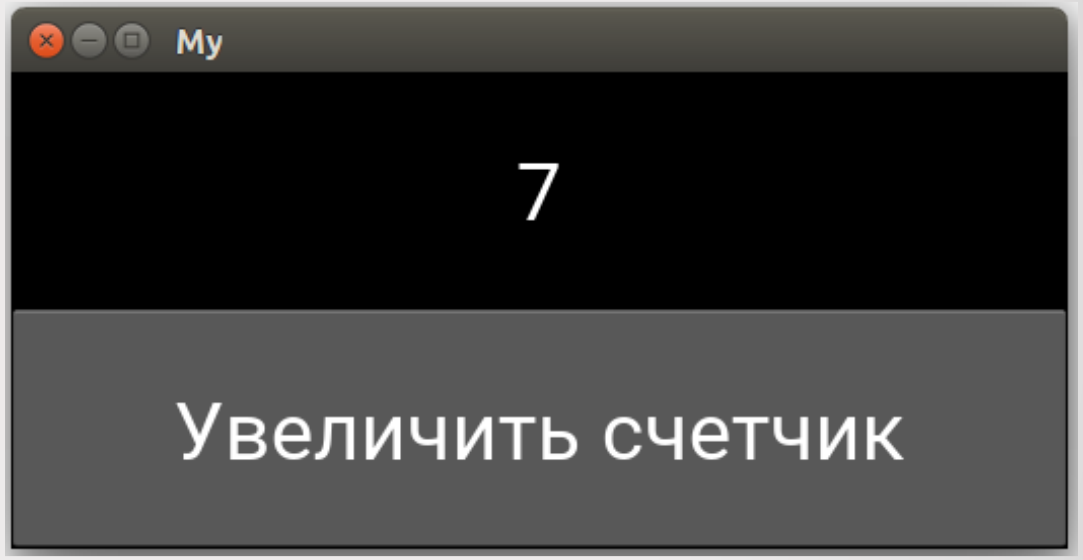
Пример приложения на **Kivy**



Пример описания интерфейса приложения

```
# main.kv
<MainWindow>:
        BoxLayout:
                size: root.size
                pos: root.pos
                orientation: 'vertical'
        Label:
                text: str(root.counter)
                font_size: 40
        Button:
                text: "Увеличить счетчик"
                on_press: root.increase()
                font_size: 40
```

Пример приложения на **Kivy**



Пример кода приложения

```
# main.py
from kivy.app import App
from kivy.uix.widget import Widget
from kivy.properties import
NumericProperty

class MainWindow(Widget):
    counter = NumericProperty(0)
    def increase(self):
        self.counter += 1

class MyApp(App): # (!) MyApp -> my.kv
    def build(self):
        return MainWindow()

MyApp().run()
```

Пример приложения на **Kivy**