Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Новосибирский национальный исследовательский государственный университет»

ФУНКЦИИ В PYTHON

СЛОВАРИ. РАБОТА СО СЛОВАРЯМИ

ФУНКЦИИ

АНОНИМНЫЕ (ЛЯМБДА) ФУНКЦИИ

Гаврилов Денис Андреевич, преподаватель кафедры СИ ФИТ НГУ

Вложенные списки.

Обход вложенных списков.

Словари.

Способы создания словарей.

Обход словарей.

Копирование словарей.

СЛОВАРИ. РАБОТА СО СЛОВАРЯМИ

СПИСКИ

Создание списка:

```
lst1 = [1, 2, 3]

lst2 = list("abcde")
```

Некоторые операции над списками:

Синтаксис	Семантика
len(s)	Длина *s*
x in s	Проверка принадлежности элемента последовательности. В новых версиях Python можно проверять принадлежность подстроки строке. Возвращает True или False
x not in s	То же, что и *not x in s*
s + s1	Конкатенация
*s*n или n*s*	Последовательность из *n* раз повторенной *s*. Если *n < 0*, возвращается пустая последовательность.
s[i]	Возвращает *i*-й элемент *s* или *len(s)+i*-й, если *i < 0*
s[i:j:d]	Срез из последовательности *s* от *i* до *j* с шагом *d*
min(s)	Наименьший элемент s
max(s)	Наибольший элемент s

ВЛОЖЕННЫЕ СПИСКИ: ОДНОМЕРНЫЙ

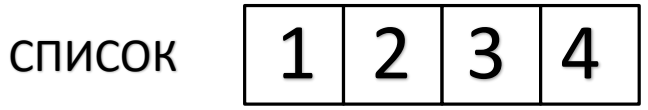
Часто в задачах приходится хранить прямоугольные таблицы с данными.

Такие таблицы называются матрицами или двумерными массивами.

В Python таблицу можно представить в виде списка, каждый элемент которого является, в свою очередь, списком.

```
lst1 = [1, 2, 3]

lst2 = list("abcde")
```



ВЛОЖЕННЫЕ СПИСКИ: ДВУМЕРНЫЙ

Создание двумерного списка:

```
n = 3
m = 4
tbl = [0] * n

for i in range(n):
    tbl[i] = [0] * m
```

Создание *n* списков в списке

СТРОКА 1:	1	2	3	4
СТРОКА 2:	1	2	3	4
СТРОКА 3:	1	2	3	4

Работа с двумерным списком:

```
for row in tbl:
    for elem in row:
        print(elem)
    print()
```

Проход по n строкам в списке
Проход по m элементам строки

ВЛОЖЕННЫЕ СПИСКИ: ДВУМЕРНЫЙ

Создание двумерного списка:

```
n = 3
m = 4
tbl = [0] * n

for i in range(n):
    tbl[i] = [0] * m
```

Создание *n* списков в списке

СТРОКА 1:	1	2	3	4
СТРОКА 2:	1	2	3	4
СТРОКА 3:	1	2	3	4

Работа с двумерным списком:

```
for i in range(n):
    for j in range(m):
        print(tbl[i][j])
    print()
```

Проход по n строкам в списке
Проход по m элементам строки

ВЛОЖЕННЫЕ СПИСКИ: ДВУМЕРНЫЙ

Пример работы со вложенными списками «вручную»:

```
a = [[1, 2, 3, 4], [5, 6], [7, 8, 9]]
s = 0

for row in a:
    for elem in row:
        s += elem
print(s)
```

Проход по «строкам» в списке
Проход по «элементам» строки

СТРОКА 1:	1	2	3	4
СТРОКА 2:	1	2	3	4
СТРОКА 3:	1	2	3	4

ВЛОЖЕННЫЕ СПИСКИ: ТРЕХМЕРНЫЙ

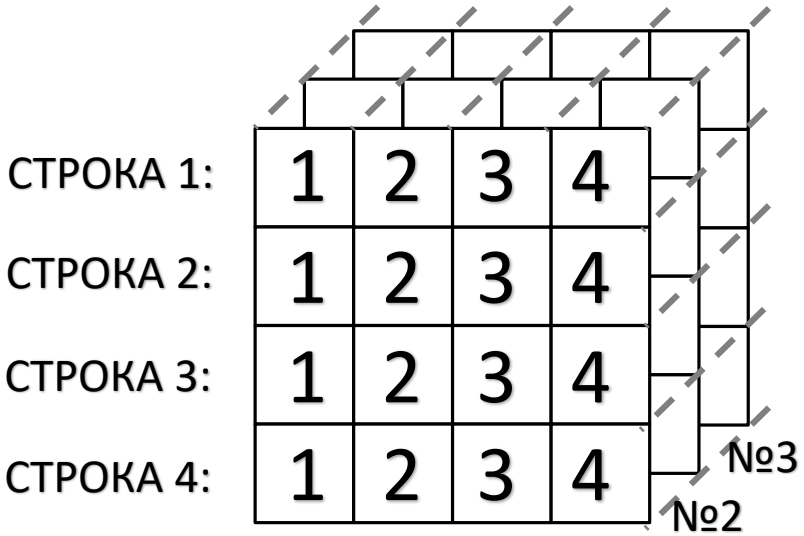
Создание трехмерного списка:

```
n = 3
m = 4
k = 4
tbl = [0] * n

for i in range(n):
    tbl[i] = [0] * m
    for j in range(m):
        tbl[i][j] = [0] * k
```

Создание *n* списков в списке…

После чего – еще по *m*
списков в каждом из *n* списков



МАТРИЦА №1

ВЛОЖЕННЫЕ СПИСКИ: ТРЕХМЕРНЫЙ

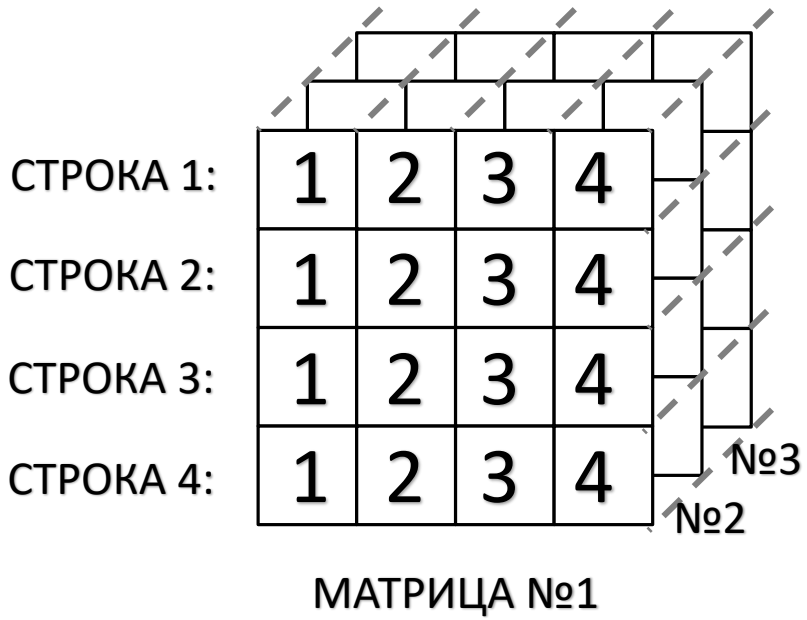
Работа с трехмерным списком:

```
for matrix in tbl:
    for row in level:
        for elem in level:
            print(elem)
        print()
    print()
```

```
# Проход по n «уровням»
# Проход по m «строкам»
# Проход по k «элементам»
```

```
for i in range(n):
    for j in range(m):
        for t in range(k):
        print(tbl[i][j][k])
    print()
```

```
# Проход по n «уровням»
# Проход по m «строкам»
# Проход по k «элементам»
```



МАТРИЦА №1

СЛОВАРИ

Словари - наборы *Объектов (значений)*, доступ к которым происходит не по индексу, как это происходит в списках, а по *ключу*.

Ключ - неизменяемый объект: число, строка или кортеж.

Ключ - слово

Объект - определение

СЛОВАРИ

- элементы словаря могут содержать объекты произвольного типа данных и иметь неограниченную степень вложенности.
- элементы в словаре располагаются в произвольном порядке.
- для получения элемента надо указать *ключ*, который использовался при сохранении *значения*.

Важно: словари относятся к *отображениям*, а не к *последовательностям*, поэтому:

к словарям не применимы функции для работы с последовательностями;

к словарям не применимы операции извлечения среза, конкатенации, повторения и др.

Однако: словари, как и списки, относятся к *изменяемым* типам данных

СОЗДАНИЕ СЛОВАРЯ

Способы создания словаря:

```
d = dict()                                  # пустой словарь d= {}
d = dict(a=1, b=2)                          # создание словаря через набор пар <Ключ>=<Значение>
d = dict({'a':1, 'b':2, 'c':3, 'd':4})      # явное создание словаря
d = dict([('a', 1), ('b', 2), ('c‘, 5)])    # создание словаря через список кортежей [(<Ключ>=<Значение>),…]
d = dict([['a', 1], ['b', 8], ['c', 12]])   # создание словаря через двумерный список [[<Ключ>=<Значение>],…]
```

РАБОТА СО СЛОВАРЯМИ

Возможно объединить два списка в список кортежей и создать словарь – при помощи функции *zip*:

```
k = ['z1', 'z2', 'z3', 'z4']
v = [1, 2, 3, 4]

d = dict(zip(k, v))


print('d=', d)
```

```
# список ключей
# список значений

# создание словаря;
# zip, в свою очередь, создает список кортежей

# Вывод: d= {'z4': 4, 'z2': 2, 'z3': 3, 'z1': 1}
```

РАБОТА СО СЛОВАРЯМИ

В случае, если списки ключей и значений не совпадают по числу элементов, *zip* использует для формирования списка кортежей кратчайший из доступных:

```
k = ['z1', 'z2', 'z3', 'z4']
v = [5, 7]

d = dict(zip(k, v))


print('d=', d)
```

```
# список ключей
# список значений

# создание словаря;
# zip, в свою очередь, создает список кортежей

# Вывод: d= {'z2': 7, 'z1': 5}
```

РАБОТА СО СЛОВАРЯМИ

Поэлементное заполнение пустого словаря («вручную»):

```
d = {}
d['a'] = 1
d['b'] = 2
d['c'] = 7

print('d=', d)

print('d['b']=', d['b'])
```

Вывод: *d= {'b': 2, 'a': 1, 'c': 7}*

Получение доступа к элементу по значению ключа [‘b‘]

РАБОТА СО СЛОВАРЯМИ

Автоматизированное создание словаря при помощи метода *fromkeys*:

```
d = dict.fromkeys(['a', 'b', 'c'])
print(d)

d = dict.fromkeys(['a', 'b', 'c'], 0)
print(d)

d = dict.fromkeys(['a', 'b', 'c'], 7)
print(d)
```

Вывод: *{'a': None, 'b': None, 'c': None}*

Вывод: *{'a': 0, 'b': 0, 'c': 0}*

Вывод: *{'b': 7, 'a': 7, 'c': 7}*

РАБОТА СО СЛОВАРЯМИ: ОБХОД

Обход словаря при помощи цикла:

```
colors = {"red": 1, “green": 2, “blue": 3}

for key in colors.keys():
    print(key)

for value in colors.values():
    print(value)

for item in colors.items():
    print(item[0], item[1])
```

Обход ключей словаря

Обход значений словаря

Обход пар ключ (item[0]) – значение (item[1])

РАБОТА СО СЛОВАРЯМИ: КОПИРОВАНИЕ

Важно: для списков и словарей **нельзя делать групповое присваивание.**

```
d1 = d2 = {'a':1, 'b':2}
```

- такой код приведет к тому, что *d1* и *d2* будут ссылаться на один и тот же словарь.

В итоге, изменение одной переменной затронет и вторую.

Вместо прямого присваивания выполняется копирование словаря.

Копия является *новым экземпляром* словаря.

РАБОТА СО СЛОВАРЯМИ: КОПИРОВАНИЕ

Создание поверхностной копии; «вручную» и при помощи метода *copy*:

```
d1 = {'a':1, 'b':2}
d2 = dict(d1)
print('d1 =', d1)
print('d2 =', d2)
print(d2 is d1)

d1['a'] = 15
print('d1 =‘ ,d1)
print('d2 =‘ ,d2)
```

```
d1 = {'a':1, 'b':2}
d2 = d1.copy()
print('d1 =', d1)
print('d2 =', d2)
print(d2 is d1)

d1['a'] = 15
print('d1 =', d1)
print('d2 =', d2)
```

Создание копии словаря *d1*

Вывод: *False* - *d2* и *d1* это разные объекты

d1 = {'a': 15, 'b': 2}
d2 = {'a': 1, 'b': 2}

РАБОТА СО СЛОВАРЯМИ: КОПИРОВАНИЕ

В случае, если значениями словаря являются данные *неизменяемых* типов, метода *copy* бывает достаточно;

Если же используются *изменяемые* типы данных (например, списки), следует применять метод *copy.deepcopy*:

```
d1 = {'a':1, 'b':[20,30,40]}
d2 = dict(d1)
print('d1 =', d1)
print('d2 =', d2)
print(d2 is d1)

d2[‘b'][0] = 15
print('d1 =‘ ,d1)
print('d2 =‘ ,d2)
```

Создание копии словаря *d1*

Вывод: *False - d2* и *d1* это разные объекты, НО

d1 = {'a': 1, 'b': [15,30,40]}
d2 = {'a': 1, 'b': [15,30,40]}

РАБОТА СО СЛОВАРЯМИ: КОПИРОВАНИЕ

В случае, если значениями словаря являются данные *неизменяемых* типов, метода *copy* бывает достаточно;

Если же используются *изменяемые* типы данных (например, списки), следует применять метод *copy.deepcopy*:

```
import copy

d1 = {'a':1, 'b':[20,30,40]}
d2 = copy.deepcopy(d1)
print('d1 =', d1)
print('d2 =', d2)
print(d2 is d1)

d2[‘b'][0] = 15
print('d1 =‘ ,d1)
print('d2 =‘ ,d2)
```

```
# Требуется подключение дополнительного пакета

# Создание копии словаря d1

# Вывод: False - d2 и d1 это разные объекты

# d1 = {'a': 1, 'b': [20,30,40]}
# d2 = {'a': 1, 'b': [15,30,40]}
```

Определение функции.

Имя, тело и параметры.

Необязательные параметры.

Вызов функции.

Понятие рекурсии.

Область видимости функции.

Изменяемость параметров.

ФУНКЦИИ В PYTHON

ФУНКЦИИ

Функция — сгруппированный фрагмент программного кода, к которому можно (неоднократно) обратиться из другого места программы.

Как правило, функции включают в себя фрагменты кода, ответственные за выполнение какой-то конечной задачи или подзадачи (например: сортировка списка, подсчет суммы элементов списка, проверка числа на четность и т.п.).

Использование функций позволяет писать более структурированный и компактный код.

ФУНКЦИИ

Важно:

- функция должна быть *описана* перед тем, как её можно будет *вызвать;*
- регистр в названии функции имеет значение – то есть функции с именами «calculate_summ» и «Calculate_Summ» будут рассматриваться интерпретатором как **две разные** функции;
- имя функции **не может начинаться с цифры.**

ФУНКЦИИ

def <Имя функции>([<Параметры>]):

["""Строка документирования"""]

<Тело функции>

return <Значение> - необязательная инструкция, возвращает некоторый результат (ответ) функции

Важно: концом функции считается выражение, перед которым меньшее количество пробелов; это не обязательно будет инструкция *return*.

ФУНКЦИИ

```
y = 2


def is_even_number(n):
    """Проверка заданного числа на четность"""
    if n % 2 == 0:
        return True
    else:
        return False


if is_even_number(y):
    print('yes')
else:
    print('no')
```

допустим, мы хотим проверить *y* на четность

опишем функцию с одним параметром *n*

определим тело функции:
проверка значения параметра n на четность;
в качестве «ответа» функции – «Правда» или «Ложь»
т.е. «четное» или «нечетное»

вызов функции;
передаем значение *y* в качестве параметра

ФУНКЦИИ

Примеры некоторых встроенных функций языка Python:

Синтаксис	Семантика
***bool**(x), **float**(x), **int**(x,n),…*	Преобразование к типу bool, float, int (для заданной системы счисления),…
***list**(x)*	Создание списка
***print**(x, *, sep=" ", end='\n', file=sys.stdout)*	Печать значения (обратите внимание на несколько необязательных параметров)
***input**([prompt])*	Возвращает введенную пользователем строку; «prompt» - подсказка пользователю
***len**(x)*	Возвращает число элементов в указанном объекте (его «длину»)
***abs**(x)*	Возвращает абсолютную величину (модуль числа)
***open**(file, mode='r', buffering=None, encoding=None, errors=None, newline=None, closefd=True)*	Открывает файл и возвращает соответствующий поток

НЕОБЯЗАТЕЛЬНЫЕ ПАРАМЕТРЫ

Чтобы сделать параметр *необязательным*, нужно присвоить ему начальное значение при объявлении функции.

Важно: необязательные параметры должны следовать после обязательных.

НЕОБЯЗАТЕЛЬНЫЕ ПАРАМЕТРЫ

Чтобы сделать параметр *необязательным*, нужно присвоить ему начальное значение при объявлении функции.

```
def f_sum(x, y=2):
    return x+y


v1=f_sum(5)                  # y – необязательный параметр,
print('v1=',v1)              # необязательные параметры необязательно задавать при вызове
                             # Вывод: 7 (использовано значение по умолчанию «2»)

v2=f_sum(10, 20)
print('v2=',v2)              # Вывод: 30 (необязательный параметр задан явно)
```

НЕОБЯЗАТЕЛЬНЫЕ ПАРАМЕТРЫ

При использовании необязательных параметров бывает удобно пользоваться *сопоставлением по ключам* при вызове функции:

```
def f_sum(x=2, y=2):
    return x+y


v1=f_sum(x=5)
print('v1=',v1)


v2=f_sum(y=20)
print('v2=',v2)
```

x,y – необязательные параметры,

Вывод: 7

Вывод: 22

РЕКУРСИВНЫЙ ВЫЗОВ ФУНКЦИИ

Пример рекурсивной функции:

Вычисление факториала числа «x» (вычисление x!)

```
def rec_func(n):
    if n < 1:
        return 1
    else:
        return n*rec_func(n - 1)


x = 5
print(rec_func(x), x)
```

если значение параметра n меньше 1,

возвратить значение 1;

в противном случае,

возвратить значение *n * (n - 1)!*

ОБЛАСТЬ ВИДИМОСТИ ФУНКЦИИ

```
def rec_func(n):
    y = n
    if y < 1:
        return 1
    else:
        return y*rec_func(y - 1)

x = 5
print(rec_func(x), x)
print(rec_func(x), n)
print(rec_func(x), y)
```

Переменные, объявленные внутри тела функции (включая параметры) не видны за пределами функции.

Такие переменные называются *локальные*.

Попытка печати значений переменных *n* и *y* приведет к ошибкам:

name 'n' is not defined

name ‘y' is not defined

ОБЛАСТЬ ВИДИМОСТИ ФУНКЦИИ

```
name = "Todd"


def say_hi():
    print("Hello,", name)


def say_bye():
    print("Good bye,", name)

say_hi()
say_bye()
```

Однако функция может использовать значения переменных, объявленные вне её тела.

Такие переменные по умолчанию считаются *глобальными*.

Вывод данного кода:

Hello, Todd

Good bye, Todd

ОБЛАСТЬ ВИДИМОСТИ ФУНКЦИИ

```
name = "Todd"


def say_hi():
    print("Hello", name)



def say_bye():
    name = 7
    print("Good bye", name)


say_hi()
say_bye()
```

Важно: *локальная* переменная может перекрыть *глобальную*, если их имена совпадают.

Даже если тип переменных в итоге различается.

Вывод данного кода:

Hello, Todd

Good bye, 7

ИЗМЕНЯЕМОСТЬ ПАРАМЕТРОВ

Важно: при передаче в функцию объектов *изменяемого типа данных* (список, словарь) **изменение** значений переменных **внутри** функции **изменит их значение вне функции**.

```
def list_modify(lst, n):
    lst[0] = n
    return lst


lst_ext = [1, 2, 3, 4]
print(list_modify(lst_ext, 5), lst_ext)
```

Вывод данного кода:

[5, 2, 3, 4] [5, 2, 3, 4]

Можно заметить, что внешний внешний список *lst_ext* оказался изменен после вызова функции.

ИЗМЕНЯЕМОСТЬ ПАРАМЕТРОВ

Обойти данную особенность можно при помощи создания копии списка:

```
def list_modify(lst, n):
    lst = lst[:]
    lst[0] = n
    return lst

lst_ext = [1, 2, 3, 4]
print(list_modify(lst_ext, 5), lst_ext)
```

копирование путем взятия «полного» среза

Вывод данного кода:

[5, 2, 3, 4] [1, 2, 3, 4]

Копирование создает полностью новый список, не связанный с передаваемым в функцию.

ИЗМЕНЯЕМОСТЬ ПАРАМЕТРОВ

Также возможно копирование при вызове функции, «на ходу»:

```
def list_modify(lst, n):
    lst[0] = n
    return lst

lst_ext = [1, 2, 3, 4]
print(list_modify(lst_ext[:], 5), lst_ext)
```

копирование путем взятия «полного» среза

Вывод данного кода:

[5, 2, 3, 4] [1, 2, 3, 4]

Копирование создает полностью новый список, не связанный с передаваемым в функцию.

ИЗМЕНЯЕМОСТЬ ПАРАМЕТРОВ

Если указать объект, имеющий *изменяемый тип данных*, в качестве значения по умолчанию, то этот объект **будет сохраняться** после каждого вызова функции в программе:

```
def f_keep(a=[]):
    a.append(2)
    return a


print(f_keep())
print(f_keep())
print(f_keep())
```

добавление нового элемента в конец списка

Вывод: [2]

Вывод: [2, 2]

Вывод: [2, 2, 2]

ПЕРЕОПРЕДЕЛЕНИЕ ФУНКЦИИ

```
def list_modify(lst, n):
    lst[0] = n
    return lst

def list_modify(lst):
    lst = [6, 7, 8]
    return lst

lst_a = [1, 2, 3, 4]
print(list_modify(lst_a), lst_a)
lst_b = [1, 2, 3, 4]
print(list_modify(lst_b, 5), lst_b)
```

Важно: функция может **переопределить** уже существующую, **если их имена совпадают**. При этом список параметров может отличаться, но переопределение все равно произойдет:

Вывод данного кода:

[6, 7, 8] [1, 2, 3, 4]

list_modify() takes 1 positional argument but 2 were given

Определение лямбда-функции. Параметры, тело.

Сохранение функций в переменные.

АНОНИМНЫЕ ФУНКЦИИ (ЛЯМБДА-ФУНКЦИИ)

ЛЯМБДА-ФУНКЦИИ

Lambda — это инструмент в Python – и многих других языках программирования – для вызова *анонимных функций*.

- У лямбда-функций нет имени, поэтому их и называют анонимные функции.
- В качестве значения лямбда-функция возвращает ссылку на объект-функцию, которую можно сохранить в переменной или передать в качестве параметра в другую функцию.
- Анонимный вызов функции подразумевает то, что её возможно вызвать, нигде заранее не объявляя.

ЛЯМБДА-ФУНКЦИИ

lambda [<Параметр1>[,..<параметрN>] : <Возвращаемое значение> (значения параметров)

В качестве примера рассмотрим функцию, вычисляющую площадь круга: $S = pi*r^2$

```
PI_CONST = 3.14


def circle_area_classic(radius):
  return PI_CONST *(radius**2)


print(circle_area_classic(5))
```

```
PI_CONST = 3.14


print((lambda radius: PI_CONST *(radius**2))(5))
```

СОХРАНЕНИЕ В ПЕРЕМЕННУЮ

Ссылку на функцию (в т.ч. анонимную) можно сохранить в переменную. Это бывает полезно при передаче функции в качестве параметра для другой функции:

```
def f_sum(x=2, y=2):
    return x+y

v1=f_sum

def sum_and_mult(x, y, sum_func, z):
    return z*sum_func(x,y)

print(sum_and_mult(5, 5, v1, 2))
```

```
v1=lambda x=2,y=2:x+y

def sum_and_mult(x, y, sum_func, z):
    return z*sum_func(x,y)

print(sum_and_mult(5, 5, v1, 2))
```

БЛАГОДАРЮ ЗА ВНИМАНИЕ!

Гаврилов Денис Андреевич, преподаватель кафедры СИ ФИТ НГУ