



МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»
(ДГТУ)**

Факультет	<u>Информатика и вычислительная техника</u>
Кафедра	<u>ПОВТиАС</u>
Направление	<u>09.03.04 Программная инженерия (бакалавриат)</u>
Дисциплина	<u>Динамические языки программирования</u>

КРАТКИЙ КОНСПЕКТ ЛЕКЦИЙ

1 ОСНОВЫ ЯЗЫКА PYTHON

1.1 КРАТКАЯ ИСТОРИЯ И ХАРАКТЕРИСТИКА ЯЗЫКА

Python — высокоуровневый язык программирования общего назначения, упор в котором поставлен на высокую производительность разработчика и легкую читаемость кода. Язык выделяется своей минималистичностью, но несмотря на это в стандартной библиотеке имеется большой объём полезных функций. Python поддерживает несколько парадигм программирования, в том числе структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное.

Основные особенности языка:

- динамическая типизация,
- автоматическое управление памятью,
- полная интроспекция
- механизм обработки исключений,

Разработка языка Python была начата в конце 1980-х годов сотрудником голландского института CWI Гвидо ван Россумом. Исходный текст был опубликован в феврале 1991 года в группе новостей alt.sources. Название язык получил в честь британского комедийного телешоу 1970-х «Летающий цирк Монти Пайтона».

1.1.1 ВЕРСИИ ЯЗЫКА И РАЗЛИЧИЯ МЕЖДУ НИМИ

Python2

Python 2 вышел в 2000 году. Данная версия сделала процесс разработки более быстрым, прозрачным и по сравнению с прошлыми реализациями.

В данной версии появилось большое число новых функций: циклический сборщик мусора, расширенную поддержку Unicode, списковую сборку и т.п. По мере разработки Python2 существенно расширился набор функций. Были добавлены унификация типов и классов Python (версия 2.2).

Python2.7

Python2.7 вышел 3 июля 2010 года и должен был стать последней версией python 2.x.

Главной целью python 2.7 было облегчить процесс перехода с python2.x на версию python3. Данная версия должна была обеспечить совместимость старой и новой версии. Она предоставляла улучшенные модули для python2.7. Как правило, на данный момент при упоминании Python2 имеется в виду версия Python2.7.

Разработка версии Python2.7 будет полностью прекращена в 2020 году.

Python3

Python3 был создан в конце 2008 года. Главной целью новой версии было устранение архитектурных недостатков предыдущих версий.

Наиболее заметные изменения в Python3.0: заменен оператор print на встроенную функцию, улучшилась поддержка unicode и изменен способ деления целых чисел.

Новая версия была воспринята прохладно, из-за несовместимости с Python2. Это в свою очередь заставляло пользователей выбирать между старой и привычной и новой версиями языка.

Различия

print

```
print "Hello world" # python2
print("Hello world") # python3
```

Деление чисел

В python2:

```
a = 5 / 2
print a
2
```

В python3:

```
a = 5 / 2
print(a)
2.5
```

```
b = 5 // 2
print(b)
2
```

unicode

По умолчанию в python2 используется ASCII. Для того чтобы использовать unicode в python2, необходимо использовать следующую конструкцию:

```
u"Hello, World!"
```

В python3 по умолчанию используется unicode.

1.1.2 РАБОТА С ИНТЕРПРЕТАТОРОМ

Для того чтобы открыть интерпретатор необходимо ввести следующую команду:

```
~ $ python
```

После этого можно вводить код:

```
>>> print("Hello world")
```

Чтоб закрыть интерпретатор необходимо ввести: <Ctrl+Z> в windows или <Ctrl+D> в *nix.

Результат последнего выражения находится в переменной `_`:

```
>>> 10 * 5
50
>>> _ * 10
500
```

При работе в консольном режиме результат выполнения кода будет выводиться на экран сразу же после нажатия клавиши Enter. Консольный режим будет удобен, если необходимо провести быстрый расчет или поэкспериментировать. В других случаях более удобным будет использование файлов с исходным кодом.

Для того чтобы выполнить код из файла необходимо передать интерпретатору python путь к файлу:

```
~ $ python ./main.py
```

Для того чтобы не указывать интерпретатор можно в первой строке файла указать путь к нему `#!/usr/bin/python`. Добавив права доступа на исполнение, можно выполнить код из файла можно следующим образом:

```
~ $ ./main.py
```

По умолчанию, интерпретатор считает, что код в исходниках созданный в кодировке UTF-8. Для того чтобы изменить данное поведение можно указать кодировку явно. Для этого в начале исходного файла необходимо добавить:

```
# -*- coding: encoding -*-
```

Где параметр «encoding» это название кодировки.

1.1.3 СТРУКТУРА ПРОГРАММЫ

Программа на языке python представляет собой текстовый файл с инструкциями, которые располагаются на отдельных строках. Если инструкция не является вложенной, то ее необходимо помещать на отдельной строке.

Присваивание выполняется с помощью «=». В конце каждой строки можно поставить точку с запятой, но это не является обязательным (и даже не рекомендуется). Вместо фигурных скобок для выделения кода в блок (как в си подобных языке) в python используются отступы, состоящие из четырех пробелов:

```
>>> i = 7
>>> while i < 15:
...     print(i)
...     i = i + 2
```

Для перевода конструкции на другую строку используется символ \ :

```
x = 2
if x >= 2\
and x < 4:
    print('hello world')
```

Язык python является интерпретируемым, поэтому выполняется последовательно.

Если необходимо, чтобы секция кода выполнялась, только если файл запущен непосредственно через интерпретатор (при импортировании код в файле так же выполняется), то необходимо воспользоваться переменной `__name__`:

```
def foo():
    print('I am from func.py')

if __name__ == '__main__':
    #Будет напечатано при вызове
```

```
print('I am from console')
```

Блок `if __name__ == '__main__':` можно рассматривать, как аналог функции `main()` в языке Си.

1.1.4 ОСНОВНЫЕ ТИПЫ ДАННЫХ И ОПЕРАЦИИ НАД НИМИ

- числа (int, float, complex);
- строки (str, bytes);
- кортежи (tuple);
- списки (list);
- словари (dict);
- множества (set, frozenset);
- логические (bool);
- NoneType.

Неизменяемые – int, float, complex, str, tuple. Остальные изменяемые.

int	
$x + y$	Сложение
$x - y$	Вычитание
$x * y$	Умножение
x / y	Деление
$x // y$	Получение целой части от деления
$x \% y$	Остаток от деления
$-x$	Смена знака числа
$\text{abs}(x)$	Модуль числа
$\text{divmod}(x, y)$	Пара ($x // y$, $x \% y$)
$x ** y$	Возведение в степень
$\text{pow}(x, y, z)$	x^y по модулю (если модуль задан)

Целые числа в python3, поддерживают длинную арифметику (в python2 с помощью отдельного типа).

Так же поддерживаются битовые операции.

$x y$	Побитовое <i>или</i>
$x ^ y$	Побитовое <i>исключающее или</i>
$x \& y$	Побитовое <i>и</i>

<code>x << n</code>	Битовый сдвиг влево
<code>x >> y</code>	Битовый сдвиг вправо
<code>~x</code>	Инверсия битов

В языке python поддерживаются различные системы счисления.

```
>>> a = int('19') # Переводим строку в число
>>> bin(19)
'0b10011'
>>> oct(19)
'0o23'
>>> hex(19)
'0x13'
>>> 0b10011
19
>>> int('10011', 2)
19
>>> int('0b10011', 2)
19
```

float

Вещественные числа поддерживают все операции целого типа. Однако вещественные числа неточны:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
0.9999999999999999
```

Если необходима высокая точность необходимо использовать Decimal и Fraction. Также float не поддерживает длинную арифметику.

complex

В Python встроены также и комплексные числа. Для работы с комплексными числами можно использовать модуль math.

```
>>> x = complex(1, 2)
>>> print(x)
(1+2j)
>>> print(x.conjugate()) # Сопряжённое число
(1-2j)
>>> print(x.imag) # Мнимая часть
2.0
>>> print(x.real) # Действительная часть
1.0
```

str

Строки в Python – это упорядоченные последовательности символов, которые используются для того, чтобы хранить текстовую информацию.

Для того, чтобы создать строку необходимо присвоить переменной текст, поместив его либо в одинарные, либо в двойные кавычки:

```
s = 'spam'
```

В python существуют экранированные последовательности:

\n	Перевод строки
\a	Звонок
\b	Забой
\f	Перевод страницы
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\N{id}	Идентификатор ID базы данных Юникода
\uhhhh	16-битовый символ Юникода в 16-ричном представлении
\Uhhhh...	32-битовый символ Юникода в 32-ричном представлении
\xhh	16-ричное значение символа
\ooo	8-ричное значение символа
\0	Символ Null (не является признаком конца строки)

Многострочная строка:

```
>>> c = '''это очень большая
... строка, многострочный
... блок текста'''
>>> c
```

Отключение экранирования:

```
s = r'C:\newt.txt'
```

Конкатенация:

```
>>> s1 = 'hello'
>>> s2 = 'world'
>>> print(s1 + s2)
' hello world'
```

Дублирование строки:

```
>>> print('ab' * 3)
ababab
```

Длина строки:

```
>>> len('spam')
4
```

Доступ по индексу:

```
>>> S = 'spam'
>>> S[0]
's'
>>> S[2]
'a'
>>> S[-2]
'a'
```

Методы строк:

s.split(символ) - разбиение строки по разделителю.

s.find(str, [start],[end]) - поиск подстроки в строке

s.startswith(str) - проверяет начинается ли строка s с шаблона str

s.endswith(str) – проверяет заканчивается ли строка s шаблоном str

s.join(list) – объединяет элементы списка в строку с разделителем s

s.capitalize() - переводит первый символ строки в верхний регистр, а все остальные в нижний

s.lower() - переводит строку в нижний регистр

s.format(*args, **kwargs) - форматирование строки

s.rstrip([chars]) - удаление пробельных символов в конце строки

s.strip([chars]) - удаление пробельных символов в начале и в конце строки

list

Списки в Python - упорядоченные изменяемые последовательности объектов, которые имеют произвольный тип.

```
>>> list('список')
['с', 'п', 'и', 'с', 'о', 'к']
>>> s = [] # Пустой список
>>> l = ['s', 'p', ['isok'], 2]
>>> s
[]
>>> l
['s', 'p', ['isok'], 2]
```

Методы списка

list.append(x)	Добавляет элемент в конец списка
list.extend(L)	Добавляет все элементы списка L

<code>list.insert(i, x)</code>	Вставляет на i-ый элемент x
<code>list.remove(x)</code>	Удаляет первый элемент в списке, имеющий значение x. Генерирует исключение, если элемент не найден.
<code>list.pop([i])</code>	Удаляет i-ый элемент и возвращает его. Если индекс не указан, то удаляется последний элемент
<code>list.index(x, [start [, end]])</code>	Возвращает индекс первого элемента со значением x
<code>list.count(x)</code>	Возвращает количество элементов со значением x
<code>list.sort([key=функция])</code>	Сортирует список на основе функции
<code>list.reverse()</code>	Разворачивает список
<code>list.copy()</code>	Копирует список
<code>list.clear()</code>	Очищает список

Методы списка изменяют сам список (в отличие от строк).

tuple

Tuple (кортеж) - неизменяемый список.

Создание кортежа:

```
>>> a = tuple() # С помощью встроенной функции tuple()
>>> a
()
>>> a = () # С помощью литерала кортежа
>>> a
()
>>>
>>> a = ('s', )
>>> a
('s',)
>>> a = 's',
>>> a
('s',)
>>> a = tuple('hello')
>>> a
('h', 'e', 'l', 'l', 'o',)
```

Кортеж поддерживает все изменяющие список операции list-a.

dict

Словари в Python – это контейнер содержащий произвольное количество пар ключ-значение.

Создание словаря:

```

>>> d = {}
>>> d
{}
>>> d = {'dict1': 1, 'dict2': 2}
>>> d
{'dict1': 1, 'dict2': 2}
>>> d = dict(short='dict', long='dictionary')
>>> d
{'short': 'dict', 'long': 'dictionary'}
>>> d = dict([(1, 1), (2, 4)])
>>> d
{1: 1, 2: 4}
>>> d = dict.fromkeys(['a', 'b'])
>>> d
{'a': None, 'b': None}
>>> d = dict.fromkeys(['a', 'b'], 20)
>>> d
{'a': 20, 'b': 20}

```

Методы словаря

dict.clear() - очищает словарь.

dict.copy() - возвращает копию словаря.

classmethod dict.fromkeys(seq[, value]) - создает словарь с ключами из seq и значением value (по умолчанию None).

dict.get(key[, default]) - возвращает значение ключа (None по умолчанию)

dict.items() - возвращает пары (ключ, значение).

dict.keys() - возвращает все ключи.

dict.pop(key[, default]) - удаляет ключ и возвращает значение.

dict.popitem() - извлекает пару (ключ, значение).

dict.setdefault(key[, default]) - возвращает значение ключа (если нет, то создает и возвращает).

dict.update([other]) - обновляет словарь.

dict.values() - возвращает все значения.

set

Множество в python – это множество уникальных элементов.

Создание множества:

```

>>> a = set()
>>> a
set()
>>> a = set('abbc')
>>> a

```

```
{ 'a', 'b', 'c' }
```

Операции над множествами

len(s) - количество элементов в множестве.

x in s - наличие x в s.

set.isdisjoint(other) - истина, если set и other нет пересекающихся элементов.

set.issubset(other) или **set <= other** - элементы set входят в other.

set.issuperset(other) или **set >= other** - элементы set входят в other.

set.union(other, other2) - объединение множеств.

set.intersection(other, other2) - пересечение.

set.difference(other, other2) – уникальные элементы множества other.

set.copy() - копия.

set.update(other, ...) - объединение.

set.intersection_update(other, ...) - пересечение.

set.difference_update(other, ...) - вычитание.

set.add(elem) - добавление элемента.

set.remove(elem) - удаление элемента.

set.discard(elem) - удаляет элемент, если он находится в множестве.

set.pop() - извлекает первый элемент (рандомный).

set.clear() – удаление всех элементов.

None

None – ничего.

```
a = None
```

Используется так же, как и null во многих других языках программирования.

Чтобы определить является ли переменная равная None необходимо использовать is:

```
>>> a is None
True
```

1.1.5 ЛИНЕЙНЫЕ АЛГОРИТМЫ

Для ввода данных в python необходимо использовать функцию input().

```
>>> a = input()
```

В результате переменная «a» будет строкой.

Для вывода данных на экран необходимо использовать функция `print()`, которая принимает неограниченное количество параметров.

```
>>> print('hello world')
hello world
```

1.2 ОСНОВНЫЕ ОПЕРАТОРЫ ЯЗЫКА PYTHON

1.2.1 ВЕТВЛЕНИЕ И ОПЕРАТОР ВЫБОРА

if

Конструкция `if-elif-else` - основной инструмент ветвления в Python.

```
if test1:
    state1
elif test2:
    state2
else:
    state3
```

```
>>> if 1:
...     print('true')
... else:
...     print('false')
...
true
```

Конструкция с несколькими `elif` может служить альтернативой конструкции `switch - case` в других языках программирования.

Оператор выбора

Оператор выбора является альтернативой тернарному оператору в других языках программирования.

```
A = B if C else E
```

```
>>> A = 't' if 'spam' else 'f'
>>> A
't'
```

1.2.2 ЦИКЛИЧЕСКИЕ АЛГОРИТМЫ

Оператор while

While - один из самых универсальных циклов в Python. Данный цикл выполняет код в теле цикла до тех пор, пока условие цикла истинно.

```
>>> i = 7
>>> while i < 15:
...     print(i)
...     i = i + 2
...
7
9
11
13
```

Оператор for

Цикл for является менее универсальным, но он гораздо быстрее цикла while. Этот цикл проходится по любому итерируемому объекту (str или list).

```
>>> for i in 'hello world':
...     print(i * 2, end='')
...
hheellllloo  wwoorrlldd
```

Операторы break и continue

Оператор continue начинает следующий проход цикла, не доходя до конца тела цикла.

```
>>> for i in 'hello world':
...     if i == 'o':
...         continue
...     print(i * 2, end='')
...
hheelllll  wwrrlldd
```

Оператор break немедленно останавливает цикл.

```
>>> for i in 'hello world':
...     if i == 'o':
...         break
...     print(i)
...
Hell
```

Код внутри блока else выполнится только, если цикл прервался без помощи break.

```
>>> for i in 'hello world':
```

```
...     if i == 'c':
...         break
... else:
...     print('Символа с в строке нет')
...
Символа с в строке нет
```

Вложенные циклы

Синтаксис вложенных циклов:

```
1 for переменная in последовательность:
2     for переменная in последовательность:
3         действие (я)
4     действие (я)
```

Можно использовать любой тип цикла в другом типе цикла. Например – цикл `for` можно вложить в цикл `while` и наоборот.

1.3 ОПИСАНИЕ ФУНКЦИЙ В ЯЗЫКЕ PYTHON

1.3.1 СИНТАКСИС ОПИСАНИЯ И ДОКУМЕНТИРОВАНИЕ ФУНКЦИИ

Существует несколько правил при создании функций в Python.

- блок функции должен начинаться с ключевого слова **def**, после которого следуют название функции и круглые скобки;
- любые аргументы функции должны находиться внутри круглых скобок;
- после скобок с аргументами идет двоеточие и с новой строки с отступом начинается тело функции.

```
def function(a):
    print(a)
```

Для вызова функции необходимо ввести имя функции и далее в скобках указать аргументы:

```
my_function("Hello")
```

Так же функция может возвращать результат:

```
def function(a):
    return a

print(function('hello'))
```

Для документирования функций используется следующий синтаксис:

```
def function(a, b):
    """function(a, b) -> list"""
```

Многострочное документирование состоит из однострочной строки документации. Далее идет пустая строка. В конце как правильно указывается

подробное описание. Первая строка может находиться на той же строке, где находятся открывающие кавычки, или же на следующей строке. Отступы в документации должны быть такие, какие имеют кавычки на первой строке.

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)

    """
    if imag == 0.0 and real == 0.0: return complex_zero
```

1.3.2 ПАРАМЕТРЫ ФУНКЦИИ

Порядок описания

Функции могут принимать любое количество аргументов.

```
>>> def funct(a, b, c):
...     return a + b + c
...
>>> funct(1, 2, 3) # a = 1, b = 2, c = 3
6
```

Вначале устанавливаются обязательные аргументы, далее аргументы, имеющие значения по умолчанию.

Значения по умолчанию

Значения по умолчанию ставятся после обязательных аргументов:

```
>>> def funct(a, b, c=2):
...     return a + b + c
```

Сверточные параметры

Если необходимо, чтобы функция принимала переменное количество позиционных аргументов, тогда перед именем ставится *:

```
>>> def func(*args):
...     return args
...
>>> func(1, 2, 3, 'abc')
(1, 2, 3, 'abc')
>>> func()
()
>>> func(1)
```

```
(1,)
```

args в данном случае - это кортеж из переданных аргументов функции. С данной переменной можно работать как с кортежем.

Если необходимо, чтобы функция принимала переменное количество именованных аргументов, тогда перед именем ставится **:

```
>>> def func(**kwargs):  
...     return kwargs  
...  
>>> func(a=1, b=2, c=3)  
{'a': 1, 'c': 3, 'b': 2}  
>>> func()  
{}  
>>> func(a='python')  
{'a': 'python'}
```

kwargs в данном случае - это словарь из переданных аргументов функции.

2 ВСТРОЕННЫЕ ТИПЫ ДАННЫХ

В составе языка Python имеются некоторые стандартные типы и структуры данных. На основе таких типов и структур возможно создание более сложных конструкций.

2.1 СТРОКИ

В языке Python строки являются не способными к мутации объектами, предоставляющими доступ в виде последовательности символов.

Строковые константы могут определяться одинарными или двойными кавычками. Для экранирования спецсимволов используется обратный слеш, как и в языке C. Для удобного составления многострочных констант предусмотрены тройные кавычки.

```
>>> "some string"
'some string'
>>> 'some string'
'some string'
>>> """some
... multiline
... string
... constant"""
'some\nmultiline\nstring\nconstant'
>>> 'or like \
... that \
... without \
... new lines'
'or like that without new lines'
```

Склеивание строк производится с помощью оператора «+», а размножение оператором «*»

```
>>> "some" + " " + "string"
'some string'
>>> "some " * 3 + "string"
'some some some string'
```

Отключение механизма экранирования производится с помощью префикса `r` перед определением строки.

```
>>> r'C:\n\some\path'
'C:\\n\\some\\path'
```

Индексация

Доступ к каждому символу может осуществляться с помощью индекса.

```
>>> str = "Some strange string"
>>> str[3]
```

```
'e'
```

Для получения фрагмента строки используется оператор среза. Первый параметр оператора является индексом начала фрагмента, второй параметр обозначает правый граничный индекс, третий в свою очередь является шагом среза.

```
>>> str = "Some strange string"
>>> str[5:12]
'strange'
>>> str[-8:-1]
'e strin'
>>> str[5:12:2]
'srne'
```

В случае, когда первый параметр не указан, индекс 0 считается началом среза. Когда не указан второй параметр, граничным индексом является длина строки. Отсутствие третьего параметра приводит к принятию шага равному 1 символу.

```
>>> str = "Some strange string"
>>> str[2:]
'me strange string'
>>> str[:4]
'Some'
>>> str[::2]
'Sm tag tig'
```

Для определения длины строки используется функция `len()`:

```
>>> len("Hello")
5
```

Кодировки

В 3 версии языка в отличие от 2, тип `str` хранит данные в Unicode. Для преобразования текста в другую кодировку используется метод `encode`, параметром которого является идентификатор целевой кодировки. Результатом работы такого преобразования является значение типа `bytes`, содержащее закодированные в целевой кодировке символы.

```
>>> "string".encode('ascii')
b'string'
>>> "string".encode('cp1251')
b'string'
>>> 'привет'.encode('cp1251')
b'\xef\x0\x08\xe2\xe5\xf2'
```

Обратное преобразование производится посредством вызова метода `decode` на байтовой строке.

```
>>> b'\xef\x0\x08\xe2\xe5\xf2'.decode('cp1251')
'привет'
```

2.2 СПИСКИ

В Python списки являются изменяемыми коллекциями произвольных объектов.

2.2.1 СПИСКОВЫЕ КОНСТАНТЫ, ИНДЕКСИРОВАНИЕ СПИСКОВ И ИХ ПЕРЕБОР, ПРИНАДЛЕЖНОСТЬ ЭЛЕМЕНТА СПИСКУ

Создание списков возможно, как статически, так и динамически. Примером статического создания списка может являться назначение переменной списковой константы:

```
my_list = [1,2,3,4,5]
```

Списки так же, как и строки индексируются, начиная с 0. Так же возможно использование отрицательных индексов для удобного доступа с конца списка.

```
>>> my_list[1]
2
>>> my_list[-1]
5
```

В случае указания индекса выходящего за границы списка, пользователь получает ошибку:

```
>>> my_list[-10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Перебор списков осуществляется с помощью цикла for:

```
>>> for elem in my_list:
...     print(elem)
...
1
2
3
4
5
```

Чтобы проверить принадлежность элемента списку используется конструкция in:

```
>>> 2 in my_list
True
>>> 7 in my_list
False
>>> 0 in my_list
False
```

2.2.2 ДОБАВЛЕНИЕ/ИЗМЕНЕНИЕ/УДАЛЕНИЕ ЭЛЕМЕНТА СПИСКА, СРЕЗЫ

Добавление элементов

Для добавления элемента в список может использоваться несколько методов. Метод `append` используется для добавления элемента в конец:

```
>>> my_list.append('hello')
>>> my_list
[1, 2, 3, 4, 5, 'hello']
```

Метод `extend` позволяет соединять списки:

```
>>> my_list = [1,2,3,4,5]
>>> my_list.extend(["some", "strange", "list"])
>>> my_list
[1, 2, 3, 4, 5, 'some', 'strange', 'list']
```

Метод `insert` вставляет элемент в заданную позицию. Первым аргументом метод `insert` принимает индекс вставки. С помощью `insert(len(a), x)` можно имитировать `a.append(x)`.

```
>>> my_list = [1,2,3,4,5]
>>> my_list.insert(0, 100)
>>> my_list
[100, 1, 2, 3, 4, 5]
>>> my_list.insert(3, 'inserted')
>>> my_list
[100, 1, 2, 'inserted', 3, 4, 5]
```

Так же для добавления элементов в конец или начало списка можно использовать оператор `+` и `+=`.

```
>>> my_list = [1,2,3,4,5]
>>> my_list+= [6]
>>> my_list
[1, 2, 3, 4, 5, 6]
>>> my_list = my_list + [7, 8]
>>> my_list
[1, 2, 3, 4, 5, 6, 7, 8]
>>> my_list = [0] + my_list
>>> my_list
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Существует возможность размножения элементов списка посредством операций ``*`` и ``*=``:

```
>>> my_list=[1,2,3,4,5]
>>> my_list*=2
>>> my_list
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

Изменение элементов

Для изменения элемента списка следует обращаться непосредственно к элементу по его индексу.

```
>>> my_list = [1,2,3,4,5]
>>> my_list[2] = 5
>>> my_list
[1, 2, 5, 4, 5]
```

Так же существует возможность замены фрагмента списка:

```
>>> my_list = [1,2,3,4,5]
>>> my_list[1:4] = ['foo', 'bar']
>>> my_list
[1, 'foo', 'bar', 5]
```

Удаление элементов

Удаление элементов из списка по индексам производится с помощью встроенной функции `del`, аргументом которой является элемент или фрагмент элементов списка:

```
>>> my_list = [1,2,3,4,5]
>>> del(my_list[1])
>>> my_list
[1, 3, 4, 5]
>>> del(my_list[1:3])
>>> my_list
[1, 5]
```

Так же удаление можно производить с использованием метода `remove`, аргументом которого является целевой элемент. Производится удаление первого встреченного элемента со значением, соответствующим аргументу:

```
>>> my_list = [1,2,3,4,5]
>>> my_list.remove(2)
>>> my_list
[1, 3, 4, 5]
```

Существует возможность удаления *i*-того элемента методом `pop`. В случае, если индекс не указан, удаляется последний элемент:

```
>>> my_list = [1,2,3,4,5]
>>> my_list.pop()
5
>>> my_list
[1, 2, 3, 4]
>>> my_list.pop(2)
3
>>> my_list
[1, 2, 4]
```

Полностью очистить список возможно с помощью метода `clear`:

```
>>> my_list = [1,2,3,4,5]
>>> my_list.clear()
>>> my_list
[]
```

2.2.3 КОНСТРУКТОРЫ СПИСКОВ, ВЛОЖЕННЫЕ СПИСКИ

С помощью конструктора `list` существует возможность создания пустого списка или преобразование перечислимого типа в список. Для создания списка последовательных чисел, например, можно использовать функцию генератор `range`, преобразовав результат ее работы в список с помощью `list()` :

```
>>> list()
[]
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Для динамического создания списка может использоваться так называемые `list comprehensions`. Для создания списка таким образом необходимо использовать цикл `for` по некоторому перечислимому типу вложенный в квадратные скобки. В данном случае на каждом элементе из `range(5)` производится размножение строки «h» :

```
>>> ["h"*i for i in range(5)]
['', 'h', 'hh', 'hhh', 'hhhh']
```

Списки могут иметь уровень вложенности. Так один список способен выступать элементом другого. Причем уровень вложенности может быть сколь угодно большим.

```
>>> [['h'],1]
[['h'], 1]
>>> [[[[1], 2], 3], 4]
[[[[1], 2], 3], 4]
>>> [[[[[[[[[1], 2], 3], 4], 5], 6], 7], 8], 9]
[[[[[[[[[1], 2], 3], 4], 5], 6], 7], 8], 9]
```

Так же динамическое создание списков поддерживает использование уровней вложенности:

```
>>> [[i*j for i in range(3)] for j in range(2)]
[[0, 0, 0], [0, 1, 2]]
```

Для доступа к элементу вложенного списка может использоваться последовательная индексация:

```
>>> my_list = [[1,2], 'hello']
>>> my_list[0][1]
2
```

В процессе динамического создания списка может применяться условный оператор `if`. В правой части для фильтрации элементов:

```
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]
```

В левой части для отображения элементов:

```
>>> [x if x % 2 == 0 else 'w' for x in range(10) ]
[0, 'w', 2, 'w', 4, 'w', 6, 'w', 8, 'w']
```

2.3 ПРОЧИЕ СТРУКТУРЫ ДАННЫХ

Python как и множество современных языков программирования имеет в своем распоряжении стандартные структуры данных, такие как множества, словари и кортежи.

2.3.1 МНОЖЕСТВА

Под множествами в Python понимается объект-контейнер класса `set`, который содержит уникальные элементы в неупорядоченном виде. Объекты множества должны быть неизменяемыми.

Создание

Конструктор позволяет создавать как пустое множество без указания параметров, так и преобразовывать другие перечислимые типы к типу `set`:

```
>>> set()
set()
>>> set([7,2,1,1,6,5,7,7])
{1, 2, 5, 6, 7}
>>> set(['h', 'g', 'j', 'g'])
{'h', 'j', 'g'}
>>> set('some string')
{'g', 's', ' ', 'm', 'o', 'i', 't', 'n', 'r', 'e'}
```

Словари, как и множества в Python имеют в своем строковом представлении символы «{» и «}». Но создать таким образом пустое множество, в отличии от не пустого, не получится:

```
>>> type({})
<class 'dict'>
>>> my_set = {1,2,3,4,4}
>>> my_set
{1, 2, 3, 4}
>>> type(my_set)
<class 'set'>
```

Так же, как и списки, существует возможность динамического создания множества, с помощью специального синтаксиса.

```
>>> { x for x in 'aabbcc'}  
{ 'a', 'c', 'b' }
```

Для добавления элемента во множество используется метод `add(elem)`:

```
>>> my_set = { 'a', 'b', 'c' }  
>>> my_set.add('e')  
>>> my_set  
{ 'a', 'c', 'b', 'e' }  
>>> my_set.add('a')  
>>> my_set  
{ 'a', 'c', 'b', 'e' }
```

Для удаления метод `discard(elem)`:

```
>>> my_set = set('abcde')  
>>> my_set  
{ 'c', 'b', 'a', 'd', 'e' }  
>>> my_set.discard('a')  
>>> my_set  
{ 'c', 'b', 'd', 'e' }
```

Операции

Над множествами возможно выполнение ряда операций, таких как пересечение, вычисление разницы другие. Объединение нескольких множество производится функцией `union` или оператором «`|`»:

```
>>> { 'a', 'b', 'c' }.union({ 'a', 'd', 'e' })  
{ 'a', 'c', 'd', 'b', 'e' }  
>>> { 'a', 'b', 'c' } | { 'a', 'd', 'e' }  
{ 'a', 'c', 'd', 'b', 'e' }
```

Пересечение вычисляется функцией `intersection` или оператором «`&`»:

```
>>> { 'a', 'b', 'c' }.intersection({ 'a', 'd', 'e' })  
{ 'a' }  
>>> { 'a', 'b', 'c' } & { 'a', 'd', 'e' }  
{ 'a' }
```

Разница вычисляется функцией `difference` или оператором «`-`»:

```
>>> { 'a', 'b', 'c' }.difference({ 'a', 'd', 'e' })  
{ 'c', 'b' }  
>>> { 'a', 'b', 'c' } - { 'a', 'd', 'e' }  
{ 'c', 'b' }
```

Класс `frozenset`

В отличие от обычного множества `set`, состояние объекта замороженного множества `forzenset` не может быть изменено:

```
>>> frozen = frozenset('abcde')  
>>> frozen  
frozenset({ 'c', 'b', 'a', 'd', 'e' })
```



```
>>> frozen.add('f')
Traceback (most recent call last):
  File "<pyshell#99>", line 1, in <module>
    frozen.add('f')
AttributeError: 'frozenset' object has no attribute 'add'
>>> frozen.discard('a')
Traceback (most recent call last):
  File "<pyshell#100>", line 1, in <module>
    frozen.discard('a')
AttributeError: 'frozenset' object has no attribute 'discard'
```

Тем не менее поддержка остальных операций над множествами обеспечена в полной мере:

```
>>> frozenset('abcd') - frozenset('ab')
frozenset({'d', 'c'})
>>> frozenset('abcd') - set('ab')
frozenset({'d', 'c'})
>>> set('abcd') & frozenset('ab')
{'a', 'b'}
```

2.3.2 СЛОВАРИ

Словарь в языке Python представляют собой коллекцию неупорядоченных элементов доступных по ключу.

Создание

Для создания словаря может применяться несколько различных подходов. Статическое создание словаря выполняется посредством указания значений между открывающейся и закрывающейся фигурной скобкой в формате «ключ:значение». Так же может использоваться конструктор, принимающий коллекцию кортежей форматом «(ключ, значение)».

```
>>> {}
{}
>>> dict()
{}
>>> {'name': 'jack', 'age': '25'}
{'name': 'jack', 'age': '25'}
>>> dict([('name', 'jack'), ('age', 25)])
{'name': 'jack', 'age': 25}
```

Так же возможно создание словаря по ключам:

```
>>> dict.fromkeys(['name', 'age'])
{'name': None, 'age': None}
>>> dict.fromkeys(['name', 'age'], 'unresolved')
{'name': 'unresolved', 'age': 'unresolved'}
```

Динамическое создание производится посредством синтаксиса схожего с синтаксисом динамического создания списков. Отличие заключается в том, что в качестве возвращаемого значения необходимо указывать ключ и значение.

```
>>> {'key' + str(i): i for i in range(5)}  
{'key0': 0, 'key1': 1, 'key2': 2, 'key3': 3, 'key4': 4}
```

Операции

Получение значения производится по ключу. В случае если словарь не содержит указанного ключа возвращается ошибка.

```
>>> my_dict['address']  
Traceback (most recent call last):  
  File "<pyshell#5>", line 1, in <module>  
    my_dict['address']  
KeyError: 'address'
```

Добавление элемента, а также обновление уже существующего производится с помощью обращения по ключу:

```
>>> my_dict['address'] = 'some city of some country'  
>>> my_dict['address']  
'some city of some country'  
>>> my_dict['name'] = 'ray'  
>>> my_dict  
{'name': 'ray', 'age': 25, 'address': 'some city of some country'}
```

Для получения коллекции ключей используется функция `keys`, коллекции значений – `values`, коллекции элементов – `items`.

```
>>> my_dict = {'name': 'jack', 'age': 25}  
>>> my_dict.keys()  
dict_keys(['name', 'age'])  
>>> my_dict.values()  
dict_values(['jack', 25])  
>>> my_dict.items()  
dict_items([('name', 'jack'), ('age', 25)])
```

2.3.3 КОРТЕЖИ

Кортеж представляет собой неизменяемую коллекцию элементов по типу список.

Преимуществом по сравнению со списком можно считать возможность использования в качестве ключа словаря, элемента множества. Так же дополнительным преимуществом является меньший размер, занимаемый в памяти.

Создание

Создать кортеж можно с помощью конструктора или используя специальный синтаксис определения кортежа:

```
>>> tuple()
()
>>> tuple([1,2,3])
(1, 2, 3)
>>> ()
()
>>> (1,2,3,)
(1, 2, 3)
```

Для создания кортежа, содержащего один элемент, обязательно необходимо указывать запятую после указания элемента. Дело в том, что скобки в случае одиночного элемента определяются как операция взятия в скобки.

```
>>> (1)
1
>>> (1,)
(1,)
```

Кортежи поддерживают все операции над списками, которые не меняют сам список.

Существует возможность прозрачного возвращения множественного результата из функции, а также возможность отображения кортежа на переменные:

```
>>> def func(a, b):
    return b, a

>>> func(1, 2)
(2, 1)
>>> first, second = func(1, 2)
>>> first
2
>>> second
1
```

3 ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В PYTHON

3.1 ОСНОВЫ ООП

3.1.1 ПОНЯТИЯ КЛАССА, ОБЪЕКТА, АТТРИБУТА, МЕТОДА

Объектно-ориентированное программирование (ООП) — парадигма программирования, в которой основными понятиями являются объекты и классы. Класс — это описание устройства объектов. Объект — это экземпляр класса. Например, `str` — это класс, а «hello world» — это объект.

Класс может иметь переменные (геометрическая фигура может иметь цвет, ширину, высоту). Эти переменные называются атрибутами. Так же классы могут иметь методы. Методы — это функции, которые принадлежат классу или объекту и служат для изменения состояния объекта или для получения данных. Примером метода, может послужить получение площади геометрической фигуры.

В python всё является объектами (`int`, `str`, `set`, `function` и т.д.).

3.1.2 НАСЛЕДОВАНИЕ, ИНКАПСУЛЯЦИЯ, ПОЛИМОРФИЗМ

Наследование

Наследование — это возможность одному классу (дочернему) получить атрибуты и методы другого (родительского) класса, причем некоторые из них могут быть заменены. Наследование может быть использовано как механизм для повторного использования кода. С помощью наследования можно независимо расширять программное обеспечение через открытые классы и интерфейсы.

Пример наследования: базовый класс — «сотрудник предприятия», классы наследники — «менеджер», «водитель», «уборщик». Данные классы объединяет способность «выполнять работу» и каждый из них имеет «должность».

Инкапсуляция

Инкапсуляция — это специальный механизм, который позволяет защитить данные и код от внешнего воздействия (путём наследования или доступа непосредственно к атрибутам).

Атрибуты и методы внутри объекта могут быть защищенными (`private`). Доступ к защищенным данным имеет лишь объект обладающий ими. Таким образом, защищенные данные скрыты для тех частей программы, которые существуют вне объекта.

Так же данные могут быть открытыми. Доступ к таким данным будут открыты для других частей программы. Часто открытые данные используются для того, чтобы сделать возможным контролируемый доступ к приватным атрибутам или методам объекта.

Полиморфизм

Полиморфизм – это механизм, который дает возможность одно и то же имя метода использовать для решения технически разных задач. Целью полиморфизма, применительно к ООП, является использование одного имени метода для задания разного поведения. Например, в языке Си, в котором полиморфизм реализован слабо, для поиска абсолютной величины числа существуют три различные функции: `abs()`, `labs()` и `fabs()`.

3.2 ОПИСАНИЕ КЛАССОВ В PYTHON

3.2.1 СИНТАКСИС ОПИСАНИЯ И НАСЛЕДОВАНИЕ КЛАССОВ

Создание классов в python

```
>>> # Пример простейшего класса, который ничего не делает
... class A:
...     pass
>>> a = A() # создание экземпляра класса A
>>> b = A()
>>> a.arg = 2 # у экземпляра a появился атрибут arg, равный 1
>>> b.arg = 3 # у экземпляра b - атрибут arg, равный 2
>>> print(a.arg)
2
>>> print(b.arg)
3
>>> c = A()
>>> print(c.arg) # у экземпляра c нет arg
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'arg'
>>> class B:
...     arg = 'hello'
...
...     def g(self):
...         return self.arg
...
>>> b = B()
>>> b.g()
'hello'
>>> B.g(b)
```

```
'hello'  
>>> b.arg = 'bye'  
>>> b.g()  
'bye'
```

Наследование классов в python

Чтобы сделать класс наследником другого класса необходимо после имени класса указать имя родительского класса в скобках:

```
>>> class CustomDict(dict):  
...     def get(self, key, default = 0):  
...         return super(CustomDict).get(self, key, default)
```

В данном примере переопределяется метод базового класса. Теперь значение по умолчанию у него 0, а не None. Для вызова метода базового класса используется функция `super()`, в которую передается родительский класс.

Инкапсуляция в python

В Python инкапсуляция реализована лишь на уровне соглашения между программистами о том, какие атрибуты и методы являются публичным, а какие — защищенными. Для того, чтобы сделать атрибут или метод защищенным необходимо добавить к его имени подчеркивание (атрибут будет доступен по имени).

```
class A:  
    def _private(self):  
        print("Защищенный метод!")  
  
>>> a = A()  
>>> a._private()  
Защищенный метод!
```

Если необходимо сделать невозможным доступ по имени, то необходимо использовать двойное подчеркивание.

```
>>> class B:  
...     def __private(self):  
...         print("Защищенный метод!")  
...  
>>> b = B()  
>>> b.__private()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: 'B' object has no attribute '__private'
```

Однако данный метод не дает полноценной защиты. Атрибут остаётся доступным имени `__ИмяКласса__ИмяАтрибута`.

```
>>> b._B__private()  
Защищенный метод!
```

Полиморфизм в python

Примером полиморфизма в python может послужить сложение объектов. Например сложение чисел и строк:

```
>>> 2 + 2
4
>>> "2" + "2"
'22'
```

3.2.2 КОНСТРУКТОР КЛАССА

В большинстве языков программирования существует некий метод, который вызывается при создании объекта. Такой метод имеет название «конструктор класса». В Python он имеет имя `__init__`.

```
class WithInit:
    def __init__(self, firstname, surname):
        self.firstname = firstname
        self.surname = surname

with_init = WithInit("Bob", "Ok")

print(with_init.firstname, with_init.surname)
```

На экран выведется «Bob Ok».

3.2.3 СПЕЦИАЛЬНЫЕ ФУНКЦИИ КЛАССА

Каждый класс имеет скрытые методы:

`__new__(cls[, ...])` - управляет созданием экземпляра. В качестве обязательного аргумента принимает класс и возвращает объект для последующей его передачи методу `__init__`.

`__init__(self[, ...])` - конструктор.

`__del__(self)` - вызывается сборщиком мусора при удалении объекта.

`__repr__(self)` - используется функцией `repr`; возвращает "сырые" данные, используемые для внутреннего представления объектов в python.

`__str__(self)` - возвращает строковое представление объекта.

`__bytes__(self)` - вызывается для преобразования к байтам.

`__format__(self, format_spec)` - вызывается функцией `format` и методом `format` у строк.

`__lt__(self, other)` - $x < y$ вызывает `x.__lt__(y)`.

`__le__(self, other)` - $x \leq y$ вызывает `x.__le__(y)`.

`__eq__(self, other)` - `a == b` вызывает `a.__eq__(b)`.

`__ne__(self, other)` - `a != b` вызывает `a.__ne__(b)`

`__gt__(self, other)` - `a > b` вызывает `a.__gt__(b)`.

`__ge__(self, other)` - `a ≥ b` вызывает `x.__ge__(b)`.

`__hash__(self)` - возвращает хэш-сумму объекта.

`__bool__(self)` - вызывается для преобразования в `bool`. Если этот метод не определён, то вызывается метод `__len__` (объекты с ненулевой длиной, являются истинными).

`__getattr__(self, name)` - вызывается, когда атрибут экземпляра класса не найден в обычных местах (например, у экземпляра нет метода с таким названием).

`__setattr__(self, name, value)` - назначение атрибута.

`__delattr__(self, name)` - удаление атрибута.

`__call__(self[, args...])` - вызов экземпляра класса как функции.

`__len__(self)` – возвращает длину объекта.

`__getitem__(self, key)` - доступ по индексу (или ключу).

`__setitem__(self, key, value)` - назначение элемента по индексу.

`__delitem__(self, key)` - удаляет элемент по индексу.

`__iter__(self)` - возвращает итератор.

`__reversed__(self)` - итератор из элементов, которые следуют в обратном порядке.

`__contains__(self, item)` - проверка на принадлежность элемента контейнеру.

3.3 ОБРАБОТКА ИСКЛЮЧЕНИЙ

3.3.1 РОЛЬ ИСКЛЮЧЕНИЙ В ОБРАБОТКЕ ОШИБОК

Обработка исключительных ситуаций — это механизм языков программирования, который служит для описания того как реагировать программе на ошибки и на другие проблемы, которые могут возникнуть во время выполнения программы.

Плюсы при использовании исключений особенно сильно проявляются при разработке библиотек и программных компонентов, которые ориентированы на массовое использование. В таких случаях программист скорее всего не знает, как именно должна обрабатываться исключительная ситуация. В таких случаях достаточно создать исключение, а обработчик необходимо будет реализовать пользователю библиотеки. Возможная альтернатива исключениям в данном случае — возврат кодов ошибок, которые необходимо передавать по цепочке

между различными уровнями программы, пока они не дойдут до места обработки. Данный подход загромождает код и резко снижает его читаемость.

3.3.2 ОПЕРАТОР ОБРАБОТКИ ИСКЛЮЧЕНИЙ TRY/EXCEPT

При возникновении исключений программа прерывается. Для того чтобы обработать исключения в Python есть конструкция `try..except`, которая имеет следующее определение:

```
try:
    инструкции
except [Тип_исключения]:
    инструкции
```

Код, в котором есть вероятность возникновения исключений, помещается в блок `try`. Если в этом коде генерируется исключение, то работа кода в блоке `try` останавливается, и выполнение начинается в блоке `except`.

После ключевого слова `except` можно указать, какое исключение будет обрабатываться (`TypeError`, `ValueError`).

```
try:
    value = int(input("Введите число: "))
    print("Число:", value)
except:
    print("Преобразование прошло с ошибкой ")
print("Завершение программы")
```

Если ввести «hello», то выполнится код в блоке `except` и на экран выведется «Преобразование прошло с ошибкой».

Пример с указанием конкретного исключения:

```
try:
    value = int(input("Введите число: "))
    print("Число:", number)
except ValueError:
    print("Преобразование прошло с ошибкой")
print("Завершение программы")
```

3.3.3 ОПЕРАТОР TRY/FINALLY

При обработке исключений так же можно использовать необязательный блок `finally`. Главной особенностью данного блока является то, что он выполнится в любой случае:

```
try:
    value = int(input("Введите число: "))
    print("Число:", number)
except ValueError:
    print("Преобразование прошло с ошибкой")
finally:
    print("Завершение программы")
```

Так же можно использовать блок `else`. Код в этом блоке выполняется после завершения блока `try` и если не было исключений, `return`, `continue` и `break`.

```
try:
    value = int(input("Введите число: "))
    print("Число:", number)
except ValueError:
    print("Преобразование прошло с ошибкой")
else:
    print("Завершение программы")
```

4 СТАНДАРТНАЯ БИБЛИОТЕКА ЯЗЫКА

Стандартная библиотека языка Python содержит широкое множество модулей, позволяющих в полной мере сосредоточиться на разработке, используя спектр готовых решений.

4.1 ВВОД/ВЫВОД

Для ввода данных из консоли может использоваться функция `input(msg)`, позволяющая остановить выполнение программы и запросить ввод пользователя. Для вывода данных в консоль используется функция `print(msg)`.

```
>>> def func():
    name = input('please enter your name:')
    print('user name: ' + name)

>>> func()
please enter your name:Jack
user name: Jack
```

4.1.1 ЧТЕНИЕ И ЗАПИСЬ ФАЙЛОВ, МЕТОДЫ ФАЙЛОВОГО ОБЪЕКТА

Для работы с файлами предусмотрена функция `open`, позволяющая открыть файл. После открытия необходимого файла, существует возможность работы с его содержимым. Режим открытия файл может отличаться в зависимости от второго параметра функции `open`.

Для открытия файла на чтение, функции `open` помимо пути к файлу необходимо передать параметр «r» обозначающий опцию `read` или чтение. В общем случае, в случае открытия файла без передачи какого-либо параметра, файл открывается в режиме текстового чтения по умолчанию.

```
>>> open('C:\\test\\file.txt', 'r')
<_io.TextIOWrapper name='C:\\test\\file.txt' mode='r'
encoding='cp1251'>
>>> open('C:\\test\\file.txt')
<_io.TextIOWrapper name='C:\\test\\file.txt' mode='r'
encoding='cp1251'>
```

Существуют следующие режимы открытия файла:

Режимы открытия файла

'r'	Открытие файла на чтение
'w'	Открытие файла на перезапись (если целевой файл не существует, производится его создание)
'a'	Открытие файла на добавление

'x'	Создание файла для записи (если целевой файл существует, будет выброшено исключение)
't'	Открытие файла в текстовом режиме
'b'	Открытие файла в двоичном режиме
'+'	Открытие файла на чтение и запись

Запись файлов

Для записи файла необходимо открыть файл в режиме записи. После открытия файла необходимо передать текст в метод `write`. Если открыть файл на данном этапе, в нем не будет содержаться новой информации. Для сохранения записанной информации необходимо корректно закрыть файл функцией `close()`.

```
>>> text = 'this is \ntest text'
>>> text
'this is \ntest text'
>>> file = open('C:\\test\\test.txt', 'wt')
>>> file.write(text)
18
>>> file.close()
```

Запись файла в бинарном режиме производится с использованием режима 'b':

```
>>> b_text = b'some text about files'
>>> file = open('C:\\test\\test.txt', 'wb')
>>> file.write(b_text)
21
>>> file.close()
```

Чтение файлов

Одним из способов прочесть файл является вызов метода `read` на объекте файла.

```
>>> file = open('C:\\test\\test.txt')
>>> file.read()
'this is \ntest text'
>>> file.close()
```

Для того чтобы прочесть только необходимое количество символов, можно указать их количество в качестве аргумента метода `read`.

```
>>> file = open('C:\\test\\test.txt')
>>> file.read(5)
'this '
>>> file.read(3)
'is '
>>> file.close()
```

Так же объект файла является итерируемым объектом и может быть использован соответственно:

```
>>> file = open('C:\\test\\test.txt')
>>> for i in file:
    i
'this is \n'
'test text'
>>> file.close()
```

И может быть преобразован в список:

```
>>> file = open('C:\\test\\test.txt')
>>> list(file)
['this is \n', 'test text']
>>> file.close()
```

Чтение в бинарном виде производится с использованием режима 'b'.

```
>>> file = open('C:\\test\\test.txt', 'rb')
>>> file.read()
b'this is \r\ntest text'
>>> file.close()
```

Файловый объект

Метод `readline` позволяет считать строку начиная с текущего положения каретки:

```
>>> file = open('C:\\test\\test.txt')
>>> file.readline()
'this is \n'
>>> file.close()
```

Метод `readlines`, возвращает список строк файла:

```
>>> file = open('C:\\test\\test.txt')
>>> file.readlines()
['this is \n', 'test text']
>>> file.close()
```

Метод `seek` позволяет изменять положение каретки в файле. В качестве параметра принимает индекс, на который необходимо передвинуть каретку относительно начала файла.

```
>>> file = open('C:\\test\\test.txt')
>>> file.read(5)
'this '
>>> file.seek(0)
0
>>> file.read(5)
'this '
>>> file.close()
```

Метод `tell` возвращает текущее положение каретки.

```
>>> file = open('C:\\test\\test.txt')
>>> file.seek(0)
0
>>> file.tell()
0
>>> file.read(5)
'this '
>>> file.read(2)
'is'
>>> file.tell()
7
>>> file.close()
```

Для того чтобы узнать кодировку считываемого файла можно использовать свойство `encoding`.

```
>>> file = open('C:\\test\\test.txt')
>>> file.encoding
'cp1251'
>>> file.close()
```

4.1.2 ОБРАБОТКА ДАННЫХ В ФОРМАТЕ JSON

Формат Json является текстовым форматом обмена данными, в основе которого лежат объекты javascript. В Python реализованная поддержка данного формата на уровне стандартной библиотеки. Для работы с форматом Json необходимо подключить модуль `json`.

```
import json
```

Модуль `json` имеет две основные функции, которые необходимы для работы с форматом Json: `dump` и `load`. Функция `dump` позволяет сериализовать и сохранить данные в формате json в файл.

```
>>> file = open('C:\\test\\test.json', 'w')
>>> obj = {'name': 'jack', 'age': 25, 'childs': [{'name': 'name1'}, {'name': 'name1'}]}
>>> json.dump(obj, file)
>>> file.close()
```

Метод `load` в свою очередь позволяет загрузить объекты из файла и десериализовать их.

```
>>> file = open('C:\\test\\test.json')
>>> new_obj = json.load(file)
>>> new_obj
{'name': 'jack', 'age': 25, 'childs': [{'name': 'name1'}, {'name': 'name1'}]}
```

Так же существуют методы, позволяющие выполнять сериализацию и десериализацию в памяти, используя строку вместо файла. Такие методы имеют соответствующие названия `dumps` и `loads`.

```
>>> string = json.dumps({'name': 'jack', 'age': 25, 'childs':
[{'name': 'name1'}, {'name': 'name1'}]})
>>> json.loads(string)
{'name': 'jack', 'age': 25, 'childs': [{'name': 'name1'},
{'name': 'name1'}]}
```

При использовании параметров по умолчанию методы `dump` и `dumps` формируют сжатый вид `Json`. В случае если необходимо удобное для чтения `Json` форматирование, следует изменить стандартное значение параметра `indent`. Данный параметр описывает количество отступов при форматировании, а также включает переводы строк. Как правило его устанавливают равным 4.

4.1.3 РАБОТА С ДВОИЧНЫМИ ФОРМАТАМИ ДАННЫХ

В Python существует двоичный тип данных `bytes`, позволяющий достаточно просто обращаться с двоичными форматами данных.

Строки и кодировки

Для создания `bytes` строки можно воспользоваться синтаксисом описания такой строки.

```
>>> b'this is bytes'
b'this is bytes'
```

Так же, чтобы преобразовать обычную строку в `bytes` можно использовать конструктор `bytes`.

```
>>> bytes('это строка', 'utf-8')
b'\xd1\x8d\xd1\x82\xd0\xbe
\xd1\x81\xd1\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'
```

Конструктор `bytes` позволяет преобразовывать итерируемый объект, целочисленных байтовых значений от 0 до 256.

```
>>> bytes([1, 57, 250, 1])
b'\x019\xfa\x01'
```

Если передать число `n` в конструктор `bytes`, вернется байтовая строка состоящая из `n` нулевых байтов.

Так же преобразовать строку из обычной в байтовую используя необходимую кодировку можно с помощью метода `encode`. Обратное преобразование производится с помощью метода `decode` на байтовой строке.

```
>>> encoded = 'это строка'.encode('utf-8')
>>> encoded
b'\xd1\x8d\xd1\x82\xd0\xbe
\xd1\x81\xd1\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'
>>> encoded.decode('utf-8')
'это строка'
```


	жадной версии необходимо использовать *?)
+	1 или более повторений объекта. Жадная версия (для использования не жадной версии необходимо использовать +?)
?	0 или 1 использование. Жадная версия (для использования не жадной версии необходимо использовать ??)
{n}	Ровно n повторений.
\	Экранирование спецсимволов.
[...]	Набор допустимых символов
a b	Соответствует и выражению «a» и выражению «b».
(...)	Группирует и сохраняет соответствующее содержимое.
\number	Соответствует содержимому группы с соответствующим номером
\A	Совпадает с началом текста
\d	Любая десятичная цифра. Аналогично [0-9].
\D	Любой символ кроме десятичной цифры. Аналогично [^0-9].
\s	Любой пробельный символ. Аналогично [\t\n\r\f\v].
\S	Любой не пробельный символ. Аналогично [^ \t\n\r\f\v].
\w	Любая буква, цифра, символ подчеркивания. Аналогично [a-zA-Z0-9_].
\W	Любой символ кроме букв, цифр и символа подчеркивания. Аналогично [^a-zA-Z0-9_].
\Z	Соответствует концу текста

Определить регулярное выражение в Python можно в виде строки:

```
>>> pattern = '^some lo+ng (.*)$'
```

Если в программе предполагается использование одного регулярного выражения более одного раза, существует возможность его компиляции в объект регулярного выражения, с последующим кэшированием, для оптимизации работы.

```
>>> comp_rp = re.compile(pattern)
```

Функции обработки регулярных выражений равноправно работают как со строковыми, так и со скомпилированными регулярными выражениями.

Компиляция регулярного выражения так же позволяет сохранять специфичные флаги, привязывая их к регулярному выражению.

Флаги регулярных выражений

Флаг	Описание
re.DEBUG	Выводит отладочную информацию о скомпилированном выражении.
re.I или re.IGNORECASE	Включает режим не чувствительности к регистру.
re.L или re.LOCALE	Изменяет поведение блоков \w \W \b \B \s \S на соответствующие текущей локали.

re.M или re.MULTILINE	Включает многострочный режим, где ^ - соответствует началу строки, а \$ - концу строки.
re.S или re.DOTALL	Включает режим соответствия '.' всем символам, включая символ перевода строки.
re.U или re.UNICODE	Включает поддержку юникод символов для \w, \W, \b, \B, \d, \D, \s and \S
re.X или re.VERBOSE¶	Включает режим читабельности регулярных выражений.

Для использования флагов при компиляции, необходимо указать их в качестве второго параметра, разделив бинарным оператором «|».

```
>>> re.compile(pattern, re.I | re.M)
re.compile('^some lo+ng (.*)$', re.IGNORECASE|re.MULTILINE)
```

4.2.2 ПРИМЕНЕНИЕ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ ДЛЯ ПОИСКА И ЗАМЕНЫ ДАННЫХ

Для работы с регулярными выражениями используются методы модуля re. В результате поиска, с помощью регулярных выражений, обычно возвращается объект MatchObject, который позволяет получить доступ к соответствующим данным. MatchObject. Доступ к группам осуществляется через метод group, с указанием номера или имени соответствующей группы. Нулевой группой считается вся подходящая под регулярное выражение последовательность. Функция groups возвращает кортеж групп, начиная с 1 группы.

search – выполняет поиск по регулярному выражению и возвращает первое встреченное совпадение в виде объекта MatchObject. В случае, когда совпадение не было найдено возвращает None.

```
>>> pattern = ".{2}(st.*)"
>>> matched = re.search(pattern, 'some string')
>>> matched.group(0)
'e string'
>>> matched.group(1)
'string'
>>> print(re.search(pattern, 'not found'))
None
```

match – выполняет поиск по регулярному выражению с начала строки. Возвращает объект MatchObject в случае совпадения, иначе None.

```
>>> pattern = ".{2}(st.*)"
>>> matched = re.match(pattern, 'some string')
>>> print(matched)
None
>>> pattern = ".*(st.*)"
>>> matched = re.match(pattern, 'some string')
>>> matched.groups()
```

```

('string',)
>>> matched.group(0)
'some string'
>>> matched.group(1)
'string'

```

`fullmatch` – проверяет строку на полное соответствие регулярному выражению. В случае если строка соответствует возвращает соответствующий `MatchObject`, иначе `None`.

```

>>> print(re.fullmatch('st.*g', 'string'))
<_sre.SRE_Match object; span=(0, 6), match='string'>
>>> print(re.fullmatch('st.*g', 'strings'))
None

```

`split` – разбивает текст по соответствующим регулярному выражению последовательностям. Возвращает список частей текста.

```

>>> re.split('\d', 'some1strange2text')
['some', 'strange', 'text']
>>> re.split('\d', 'some strange text')
['some strange text']

```

`findall` – возвращает все подходящие регулярному выражению строки в виде списка.

```

>>> print(re.findall('st.*?g', 'some strings are strong'))
['string', 'strong']

```

`sub` – производит замену встреченных совпадений на указанную последовательность. В т.ч. с возможностью использования групп, из текущего регулярного выражения.

```

>>> re.sub('(s[^\s]*ong)', r'replaced(\1)', 'some strings are strong')
'some strings are replaced(strong)'

```

4.3 ОРГАНИЗАЦИЯ МАТЕМАТИЧЕСКИХ ВЫЧИСЛЕНИЙ

Python является одним из языков с мощнейшей поддержкой математических вычислений и обширным набором инструментов для работы со статистикой.

4.3.1 МОДУЛЬ MATH

В стандартную библиотеку Python встроен модуль `math` для проведения математических вычислений, содержащий обширный набор функций.

Функции модуля `math`

Функция	Описание
<code>sin(r)</code>	Вычисление синуса угла (в радианах)

cos(r)	Вычисление косинуса угла (в радианах)
tan(r)	Вычисление тангенса угла (в радианах)
acos(r)	Вычисление арккосинуса угла (в радианах)
asin(r)	Вычисление арксинуса угла (в радианах)
atan(r)	Вычисление арктангенса угла (в радианах)
log(n, b)	Вычисление логарифма числа n с основанием b
log10(n)	Вычисление десятичного логарифма числа n
factorial(n)	Вычисление факториала числа n
degrees(r)	Перевод угла r из радиан в градусы
radians(g)	Перевод угла g из градусов в радианы
floor(n)	Округление числа n до наименьшего целого
ceil(n)	Округление числа n до наибольшего целого
sqrt(n)	Вычисление квадратного корня от n
pow(n,p)	Вычисление p степени от числа n

Для работы с функциями модуля `math` необходимо импортировать данный модуль.

```
>>> import math
>>> math.floor(0.0445)
0
>>> math.ceil(0.0445)
1
>>> math.sqrt(4)
2.0
>>> math.pow(10,3)
1000.0
```

4.3.2 ГЕНЕРАЦИЯ СЛУЧАЙНЫХ ЧИСЕЛ

В стандартной библиотеке Python предусмотрен модуль обеспечивающий функциональность генератора псевдослучайных чисел. Таким модулем является `random`.

Функции модуля `random`

seed(x)	Инициализирует генератор. По умолчанию используется системное время.
randrange(from, to, step)	Генерирует случайное число из последовательности.

<code>randint(a, b)</code>	Генерирует случайное целое число x в диапазоне $a \leq x \leq b$
<code>choice(seq)</code>	Производит случайную выборку элемента из последовательности.
<code>shuffle(seq)</code>	Производит перемешивание коллекции в случайном порядке. При этом коллекция должна быть изменяемой.
<code>random()</code>	Генерирует случайное число с плавающей точкой от 0 до 1
<code>uniform(a, b)</code>	Генерирует случайное число x с плавающей точкой в диапазоне $a \leq x \leq b$

Для работы с функциями модуля `random` необходимо импортировать данный модуль.

```
>>> import random
>>> random.seed('some seed')
>>> random.randint(0,100)
86
>>> random.randrange(0,100,20)
20
>>> random.choice([1,25,79])
79
>>> seq = [1,25,79]
>>> random.shuffle(seq)
>>> seq
[79, 1, 25]
>>> random.random()
0.07956026480163814
>>> random.uniform(1,5)
4.0688765650700915
```

4.3.3 БИБЛИОТЕКИ NUMPY И SCIPY

Для работы с многомерными массивами и матрицами, а также многими высокоуровневыми функциями в языке Python существует библиотека NumPy. Исходный код библиотеки NumPy находится в открытом доступе. Данная библиотека оптимизирует различные операции над матрицами и по скорости работы, а также возможностям сопоставима с Matlab. Благодаря интеграции с библиотекой SciPy появляется возможность использования готовых решений в сфере инженерных и научных расчетов. Данные библиотеки так же поддерживают хороший уровень интеграции с другими инструментами, такими как Matplotlib – для графического отображения данных, RPy – для анализа данных с помощью языка программирования R и других.

Для работы с NumPy необходимо его установить. Проще всего это сделать с помощью менеджера пакетов `pip`.

```
pip install numpy
```

В случае, если предполагается использовать множество других научных библиотек, одним из лучших способов является использование дистрибутива Anaconda с предварительно установленными и сконфигурированными пакетами.

Функции модуля numpy

array	Преобразует последовательности в n-мерную матрицу.
zeros	Создает матрицу, заполненную нулями.
ones	Создает матрицу, заполненную единицами.
eye	Создает единичную матрицу.
arange	Создает последовательность исходя из параметров start, end, step.
linspace	Создает последовательность исходя из параметров start, end, num.

Для работы с функциями модуля numpy необходимо импортировать данный модуль.

```
>>> import numpy as np
>>> np.array([1,2,3,4,5])
array([1, 2, 3, 4, 5])
>>> arr = np.array([[1,2],[3,4]])
>>> arr
array([[1, 2],
       [3, 4]])
>>> arr[0][1]
2
>>> np.array([[1,0,0],[0,1,0], [0,0,1]])
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
>>> np.zeros((3,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> np.ones((5,5))
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
>>> np.eye(4)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> np.arange(1, 100, 3)
array([ 1,  4,  7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40,
        43, 46, 49, 52, 55, 58, 61, 64, 67, 70, 73, 76, 79, 82, 85, 88,
        91, 94, 97])
>>> np.linspace(1, 100, 5)
```

```
array([ 1. , 25.75, 50.5 , 75.25, 100. ])
```

Пример работы с пакетом `scipy`:

```
>>> import scipy.misc as sm  
>>> sm.factorial([2, 2.5, 7])  
array([ 2.00000000e+00,  3.32335097e+00,  5.04000000e+03])
```