Лекция 2

Типы данных и операции

в языке Python

Строки, байты и байтовые массивы

Свойства типа данных

- Объект имеющий определенный тип обладает набором свойств. Часто для простоты говорят *тип данных имеет свойство*.
- В этой лекции мы будем иметь дело со следующими свойствами объектов:
  - свойство быть *изменяемым* (mutable) или *неизменяемым* (immutable)
  - свойство быть *перечислимым* (enumerable), также используется термин "итерируемый тип"
  - свойство быть *упорядоченным* (sequence) или, другими словами, свойство быть последовательностью
    - последовательность это итерируемый тип, дополнительно обладающий свойством упорядоченности

Категории типов данных

Встроенные типы данных

- Встроенные типы данных не нужно импортировать явно
- Среди встроенных типов данных есть как простые, так и композитные

Типы данных определяемые пользователем (user-defined)

- Класс созданный программистом становится типом данных для производимых от него объектов
- Вместе с интерпретатором поставляется стандартная библиотека языка
  - Типы данных включенные в стандартную библиотеку можно рассматривать как типы данных определенные пользователем

Встроенные типы данных

- Простые типы данных
  - *Числа*
  - *Логический тип bool* со значениями *True* и *False*
  - Тип *NoneType* имеющий единственное значение *None,*
    - None используется как индикатор отсутствия информации. Аналогичен указателю NULL в языке C.
- Композитные типы данных

Композитные типы данных

- *Коллекции* это композитные типы, которые можно рассматривать как совокупность объектов
- Последовательности символов или байт
  - *Строки* (неизменяемые последовательности символов)
  - *Байты* (неизменяемые последовательности байт)
  - *Байтовые массивы* (изменяемые последовательности байт)
- Все элементы последовательности символов или байт имеют один и тот же тип
- Другие встроенные коллекции
  - *Списки* (в других языках это массивы), литерал [ ]
  - *Кортежи* (неизменяемые списки), литерал ( )
  - *Словари* и *множества* (в других языках словарь это ассоциативный массив, хэш или map), литерал { }
- Списки, кортежи, словари и множества могут содержать элементы разных типов
- Объекты имеющие тип класса созданного пользователем также могут быть коллекциями

Литералы

- Литерал это "буквальная" запись содержимого объекта
  - Числовые литералы аналогичны литералам языка C:
    - Десятичные `45`
    - Шестнадцатеричные `0x45`
    - Восьмеричные `0o45 (в языке C 045)`
    - Двоичные `0b1101 (в языке C начиная со стандарта C++14)`
  - Текстовые литералы (строки):
    - `'abc\ndef' # литерал в одинарных кавычках`
    - `"abc\ndef" # литерал в двойных кавычках`
    - литерал в трех кавычках допускает перенос строки:

      ```
      """abc    '''abc  # Четыре литерала-строки на этом слайде содержат
      def"""    def'''  # одинаковые последовательности символов
      ```

    - литералы в одинарных кавычках и литералы в двойных кавычках имеют одинаковые свойства

Числа

- Целые числа, разрядность не ограничена
- Числа с плавающей запятой
- Числа с фиксированной запятой (класс Decimal)
- Рациональные числа или дроби (класс Fraction)
- Комплексные числа
- Встроенные функции (built-in call) позволяют осуществлять преобразование к нужному типу:
  - `a = 25 + int("28") => 53`
  - `b = str(25) + "28" => "2528"`
  - `c = 3 + int(2.8) # => 5`
- Примеры числовых литералов:
  - `123, 0x123, 0o123, 0b1101`
  - `1.23, 12., .12, 1.2e3, 1.2+3.4j, 3.4J`

Операторы и операции

- *Инструкция* это одно из элементарных действий, последовательность которых составляет программу
- *Операция* это действие над объектами или объектом, совершаемое при вычислении выражения
- *Оператор* это символ или зарезервированное слово для обозначения операции
- Большая часть операторов языка двуместные:
  - `a + b # сложение, коммутативная операция`
  - `a - b # вычитание, некоммутативная операция`
- Одноместные операторы, например:
  - `-x # унарный минус`
- Трехместный оператор `x if y else z` аналогичен оператору ?: языка C

```
значение_если_да if вопрос else значение_если_нет # Питон
вопрос ? значение_если_да : значение_если_нет     # Язык C
```

Операции над числами:

- операнды приводятся к наиболее длинному типу (как в языке C)
- сложение: оператор +
- вычитание: оператор -
- умножение: оператор *
- деление: оператор /
- остаток от деления: оператор %
- деление с округлением вниз: оператор //
  - результат округляется до ближайшего меньшего целого
  - тип результата сохраняется
  - для классического округления есть встроенная функция round()
- возведение в степень: оператор ** или функция pow(x, y)

- Модуль math содержит базовый набор математических функций: sin(), cos(), sqrt() и др.

Строки, байты и байтовые массивы

- Строка, тип **str**
  - последовательность *символов*
  - объекты *неизменяемы*
- Байты, тип **bytes**
  - последовательность *байт*
  - объекты *неизменяемы*
- Байтовый массив, тип **bytearray**
  - последовательность *байт*
  - объекты *изменяемы*
- Все три типа можно считать *коллекциями*
- Все три типа обладают свойством упорядоченности
- Объекты создаются вызовом функции, имя которой совпадает с именем типа: str() bytes() и bytearray(). Эти же функции служат для преобразования объекта к нужному типу.

Литералы-строки и литералы-байты

- Литералы-байты имеют тот же синтаксис что и литералы-строки, но:
  - имеют префикс *b* перед открывающей кавычкой
  - могут содержать только символы ASCII или числовые коды от 0 до 255 в виде Esc-последовательностей языка C
- Примеры:

```
# Литералы-строки

s = 'abcdef'
s = "abcdef"
s = '''Line 1
Line 2
Line 3
'''
s = """Line 1
Line 2
Line 3
"""
```

```
# Литералы-байты

a = b'abcdef'
a = b"abcdef"
a = b'''Line 1
Line 2
Line 3
'''
a = b"""Line 1
Line 2
Line 3
"""
```

Произвольный символ в литерале

- В литерал-строку или литерал-байты можно вставить произвольный символ Unicode
- Для вставки в строку произвольного символа Unicode его шестнадцатеричное численное значение записывается после специального префикса
  - \x - для однобайтового символа
  - \u - для двухбайтового символа
  - \U - для четырехбайтового символа
- Примеры для строк:

```
'\xac', '\u20ac', '\U0001800a' # три строки по одному символу
'\xac\u20ac\U0001800a'         # строка из трех символов
```

- Для литералов-байт допустим только префикс \x
  Пары символов \u и \U специального значения не имеют и интерпретируются "как есть"

Строки и символы

- В Питоне нет типа *символ*. Вместо единичного символа используется текстовая строка, содержащая один символ.
- В Питоне есть ряд функций и методов для работы с символами аналогичных функциям языка С. Большая часть этих функций вместо действия *для одного символа* совершает действие *для всех символов строки*.
- Функция *ord()* преобразует символ в его числовое значение в таблице Unicode. Для символов ASCII это числа от 0 до 127.
  - Функция ord() применима только к строкам единичной длины, при более длинной строке происходит ошибка

```
ord('я')  # => 1103
ord('яя') # => Ошибка
```

- Функция *chr()* преобразует число в строку с одним символом, код которого в таблице Unicode равен этому числу

```
chr(1103) # => 'я'
```

Оператор [ ]

- Оператор [ ] служит для обращения к единичному элементу или фрагменту последовательности
- Выбранный элемент или фрагмент может находиться в левой части инструкции присваивания, в этом случае происходит его модификация
- Выбор элемента по индексу, индексация начинается с нуля. Отрицательный индекс означает отсчет элементов с конца.
  - s[0], s[-1] # первый и последний элементы последовательности,
  - s[a:b] # фрагмент от символа с индексом *a* (включительно) до символа с индексом *b* (ИСКЛЮЧИТЕЛЬНО !)
  - s[a:b:c] # *c* - шаг перебора символов
    - умолчания: a = 0, b : до конца последовательности, c = 1
  - s[:] # Вся последовательность, создается *копия* последовательности s

Оператор [ ], примеры

- Выборка элемента:

```
s = 'abcdef'
s[2] => 'c'
s[-2] => 'e'
```

- Выборка фрагмента:

```
s[2:] => 'cdef'
s[:-2] # => 'abcd'
s[:] # => 'abcdef'
s[::2] # => 'ace'
```

- Модификация, к типам *str* и *bytes* неприменима:

```
a = bytearray(b'abcdef')
a[2] = ord(b'x')  # => b'abxdef'
a[2:] = b'123'    # => b'ab123'   # Размер заменяющего и заменяемого
a[:-2] = b'98'    # => b'9823'    # фрагмента могут не совпадать
a[:] = b'newtext' # => b'newtext' # Если указан шаг, совпадение
a[::2] = b'1234'  # => b'1e2t3x4' # размеров фрагментов обязательно
```

Операторы + и *

- Строки, байты и байтовые массивы можно "складывать". Результатом будет новый объект, являющийся соединением (конкатенацией) оригинальных объектов в порядке их следования. Тип результата это тип *левого* объекта.
- Примеры:

```
s = 'Long' + 'Name' # => LongName
a1 = bytearray(b'Long') + b'Name' # => bytearray(b'LongName')
a2 = b'Long' + bytearray(b'Name') # => b'LongName'
```

- Строки байты и байтовые массивы можно "умножать" на целое число. При умножении на N результатом будет новый объект, в котором содержимое оригинального объекта повторено N раз. Операция коммутативна.
- Примеры:

```
s = 'Abc' * 6 # => AbcAbcAbcAbcAbcAbc
s = 72 * '='  # => Строка из семидесяти двух символов '='
```

Создание объектов строк или байт

- Объект может быть определен литералом
- Объект может быть создан вызовом встроенной функции имя которой совпадает с именем класса объекта
- Объект может быть скопирован выделением фрагмента включающего оригинальный объект целиком
- Класс bytearray имеет метод copy()
- Объект может быть считан из файла (материал следующих лекций)

```
# Строки            # Байты              # Байтовые массивы

s1 = 'abcdef'       x1 = b'abcdef'       нет
s2 = str(s1)        x2 = bytes(x1)       a2 = bytearray(x1)
s3 = s2[:]          x3 = x2[:]           a3 = a2[:]
нет                 нет                  a4 = a3.copy()
```

Методы классов

str, bytes и bytearray

Виды методов

- *Модифицирующий метод*
  - метод производящий изменения непосредственно в объекте
- *Копирующий метод*
  - метод создающий копию объекта и уже в этой копии производящий изменения
- *Информирующий метод*
  - метод возвращает информацию о состоянии объекта или отдельных его частей. Сам объект не изменяется, так как в изменениях нет необходимости
- *Метод дающий доступ*
  - метод возвращает ссылку на объект или на его составную часть. В ходе исполнения метода объект не изменяется, но он может быть изменен впоследствии

Поиск и замена

- Приведенные ниже методы применимы к строкам, байтам и байтовым массивам
- Методы не изменяют объект
- Примеры для строк:

```
s1 = '123 456 444 654 321'
s1.count('4') # Сколько раз '4' встречается в строке s1 ?
s1.find('4') # Найти '4', результат - индекс найденного
s1.rfind('4') # Найти '4', но искать с конца
s1.index('4') # Найти '4', результат - индекс найденного
s1.rindex('4') # Найти '4', но искать с конца
s1.replace('4', '?') # Найти все '4' и заменить на '?'
s1.replace('4', '?', 3) # Найти все '4' и заменить первые 3 из них
s1.startswith('123') # Начинается ли строка с указанной подстроки ?
s1.endswith('321') # Заканчивается ли строка указанной подстрокой ?
```

- Метод [r]find возвращает -1 если подстрока не найдена
- Метод [r]index вызывает ошибку если подстрока не найдена

Поиск и замена, синтаксис

- Синтаксис методов предыдущего слайда на примере метода *find*

```
find(sub[, start[, end]])
```

  - *sub* - подстрока, поиск которой происходит
  - *start* - начальный индекс для поиска
  - *end* - конечный (не включая) индекс для поиска
- Тип параметра поиска должен соответствовать типу объекта для которого вызывается метод
- Синтаксис метода *replace*

```
replace(old, new[, count])
```

  - *old* - подстрока, поиск которой происходит
  - *new* - подстрока заменяющая найденную подстроку
  - *count* - максимальное количество замен

Оператор in

- Оператор *in* служит для проверки утверждения: *Присутствует ли элемент X в коллекции Y*
- Для строк, байт и байтовых массивов оператор *in* действует аналогично методам find() или count(), но результатом будет логическое значение
- Оператор *in* проверяет наличие в проверяемой строке как отдельных символов (байт), так и строк
- Примеры:

```
s1 = '123 456 444 654 321'
'4' in s1   # => True
'456' in s1 # => True

a1 = b'123 456 444 654 321'
b'9' in a1   # => False
b'457' in a1 # => False
```

Разделение и соединение

- Приведенные ниже методы применимы к строкам, байтам и байтовым массивам
- Методы создают новый объект или объекты. Исходный объект остается неизменным
- Примеры для строк:

```
s1 = '123 456 444 654 321'
s1.split() # Деление по пробельным символам
s1.split('5') # Деление по разделителю
s1.split(maxsplit=2) # Деление с указанием числа точек деления
s1.rsplit(maxsplit=2) # Деление с указанием точек деления справа
s1.partition('4') # Деление на три части: левая, разделитель, правая
s1.rpartition('4') # Деление на три части справа

s3 = 'Line 1\nLine 2\nLast line'
s3.splitlines() # Разделение на строки

s2 = '-'
s2.join(s1.split()) # Соединение разделенного методом split
```

Разделение и соединение, синтаксис

- Синтаксис методов

```
split(sep=None, maxsplit=-1)
rsplit(sep=None, maxsplit=-1)
partition(sep)
rpartition(sep)
join(iterable)
```

- Если в методах split и rsplit параметр *sep* не указан, разделителем будут служить подстроки целиком состоящие из пробельных символов
- В методах partition и rpartition параметр sep обязателен
- Методы split и rsplit возвращают список, методы partition и rpartition возвращают кортеж
- Параметром метода join может быть любая коллекция *все элементы* которой имеют одинаковый тип, совпадающий с типом объекта для которого метод был вызван

Методы модифицирующие объект

- Методы модифицирующие объект неприменимы к неизменяемым типам str и byte
- Примеры для bytearray

```
a1 = bytearray(b'123 456 444 654 321')
a1.clear()         # Очистить массив
a1.append(0x61)    # Добавить один байт в конец массива
a1.extend(b'AC')   # Добавить последовательность байт в конец массива
a1.insert(2, 0x62) # Вставить байт в указанной позиции
a1.pop()        # Удалить последний байт
a1.pop(3)       # Удалить байт в указанной позиции
a1.pop(-3)      # Удалить байт в указанной позиции с конца массива
a1.remove(0x41) # Удалить первый байт с указанным значением
a1.reverse()    # Изменить порядок байт в массиве на противоположный
```

- Параметр методов append, insert и remove это число в пределах от 0 до 255
- Параметр метода append это объект типа bytes / bytearray, или список / кортеж, содержащий числа в пределах от 0 до 255

Методы форматирования текста

- Приведенные ниже методы применимы к строкам, байтам и байтовым массивам
- Методы создают новый объект. Исходный объект остается неизменным
- Для байт и байтовых массивов предполагается текст в кодировке ASCII
- Методы преобразования прописных и строчных букв

```
s1 = 'text example\taNd Other words'
s1.title() # Сделать первую букву строки прописной
s1.capitalize() # Сделать первые буквы слов прописными
s1.lower() # Сделать все буквы строчными
s1.upper() # Сделать все буквы прописными
s1.swapcase() # Поменять строчные буквы на прописные и наоборот
```

Методы форматирования текста

- Заполнение поля фиксированной длины

```
s1 = 'text example\taNd Other words'
s1.ljust(36)   # Разместить текст слева
s1.center(36)  # Разместить текст в центре
s1.rjust(36)   # Разместить текст справа
'123'.zfill(6) # Расширить строку нулями слева
```

- Удаление пробельных символов

```
s2 = '   123   '
s2.strip()  # Удалить пробелы в начале и конце строки
s2.lstrip() # Удалить пробелы в начале строки
s2.rstrip() # Удалить пробелы в конце строки
s1.expandtabs(tabsize=8) # Расширить табуляции в пробелы
```

- Методы strip могут иметь параметр-строку (строку байт), в этом случае удаляются все символы, присутствующие в параметре

Информация о тексте

- Приведенные ниже методы применимы к строкам, байтам и байтовым массивам
- Методы возвращают логическое значение True если *все* символы или байты удовлетворяют условию заявленному в названии метода
- Для байт и байтовых массивов предполагается текст в кодировке ASCII
- Примеры, все методы возвращают True

```
  ' \t\f\r\n '.isspace() # пробельные символы ?
  '1234'.isdigit() # цифры ?
  't=2?'.isascii() # символы ASCII ? (версия Питона >= 3.7)
  'text'.isalpha() # буквы ?
  'txt2'.isalnum() # цифры или буквы ?
  'tex2'.islower() # если буквы, то все строчные ?
  'TEX2'.isupper() # если буквы, то все прописные ?
  'Tex2'.istitle() # начинается с прописной буквы ?
Метод isxdigit() реализован в модуле curses.ascii
```

Трансляция

- Трансляция это преобразование символов или байт по таблице формата *исходное значение -> результирующее значение*
- Метод maketrans() создает таблицу трансляции:

```
maketrans(исходные_значения, результирующие_значения)
```

  Строки таблицы создаются из элементов с одинаковыми индексами
- Метод translate() выполняет трансляцию:

```
s1 = 'Last lump'
t = str.maketrans('aLup', 'eBib')
s1.translate(t) # => 'Best limb'
```

- Методы maketrans() и translate() применимы к строкам, байтам и байтовым массивам
- Метод translate() создают новый объект. Исходный объект остается неизменным

Преобразование строк в байты

- Метод *encode*, если кодировка не задана явно, используется кодировка операционной системы

```
s = 'Home Дом' # => 'Home Дом', длина 8 символов
b = s.encode('utf-8') # Кодировка задана явно
  # => b'Home \xd0\x94\xd0\xbe\xd0\xbc', длина 11 байт
c = s.encode() # Используется кодировка ОС
  # => b'Home \xd0\x94\xd0\xbe\xd0\xbc', длина 11 байт
```

- Преобразование при создании объекта вызовом встроенной функции, *указание кодировки обязательно*

```
b = bytes('Home Дом', 'utf-8') # => b'Home \xd0\x94\xd0\xbe\xd0\xbc'
b = bytearray('Home Дом', 'utf-8') # => то же, но bytearray
b = bytearray('Home Дом') # => Вызывает ошибку
```

Преобразование байт в строки

- Метод *decode*, если кодировка не задана явно, используется кодировка операционной системы

```
b = b'Home \xd0\x94\xd0\xbe\xd0\xbc' # длина 11 байт
s = b.decode('utf-8') # => 'Home Дом', длина 8 символов
t = b.decode() # => 'Home Дом', длина 8 символов
```

- Последовательность байт не обязательно представляет собой строку в заданной кодировке. Обработка ошибок:

```
b.decode(errors='replace') # заменять неправильные символы
b.decode(errors='ignore')  # пропускать неправильные символы
```

- Преобразование при создании объекта вызовом встроенной функции

```
s = str(b'Home \xd0\x94\xd0\xbe\xd0\xbc', 'utf-8') # => 'Home Дом'
```

- Если не указать кодировку результатом будет строка, содержащая буквальный текст двоичного литерала

```
s = str(b'Home \xd0\x94\xd0\xbe\xd0\xbc')
  => "b'Home \xd0\x94\xd0\xbe\xd0\xbc'"
```

Утилита iconv

- Утилита командной строки *iconv* позволяет преобразовать текстовый файл из одной кодировки в другую
- Синтаксис команды можно узнать использовав опцию --help

```
iconv --help
```

- Пример перекодировки из кодовой таблицы Microsoft 1251 в общепринятую кодировку utf-8

```
iconv -f cp1251 -t utf8 windows_text.txt > linux_text.txt
```

  - опция -f означает *from*, опция -t означает *to*,
    Если опция не задана, используется кодировка заданная в операционной системе.
  - символ > перенаправляет вывод в файл
- Допустимые кодировки можно увидеть использовав опцию -l

```
iconv -l  # -l означает list all known character set encodings
```

Функции pack() и unpack()

- Функции pack() и unpack() предназначены для работы с двоичными (не текстовыми) данными. Функции находятся в модуле *struct*.
- Функция *pack()* возвращает объект типа bytes в который в соответствии с параметром format будут упакованы переданные ей параметры
- Некоторые символы формата:

```
b - signed char  (1 байт)        B - unsigned char  (1 байт)
h - signed short (2 байта)       H - unsigned short (2 байта)
i - signed int   (4 байта)       I - unsigned int   (4 байта)
q - signed long long (8 байт)    Q - unsigned long long (8 байт)
f - float (4 байта)              d - double (8 байт)
? - bool  (1 байт)               Ns - char[N] (N байт)
```

- Некоторые символы модификаторов формата:

```
> - числа в формате big-endian
< - числа в формате little-endian
```

Функции pack() и unpack()

- Функция *unpack()* извлекает значения величин из объекта типа bytes или bytearray в соответствии с параметром format и возвращает эти значения в виде кортежа
- Синтаксис:

```
from struct import pack, unpack
pack(format, v1, v2, ...)
unpack(format, buffer)
```

- Примеры:

```
# 260 = 1 * 256 + 4 * 1
pack('>h', 260) # => b'\x01\x04'
pack('<h', 260) # => b'\x04\x01'
pack('h', 260)  # => b'\x04\x01' - для процессоров Intel и AMD
unpack('>h', b'\x04\x01') # => (260,)
unpack('<h', b'\x01\x04') # => (260,)
unpack('<hl2s', b'\x04\x01\x03\x00\x00\x00\x61\x62') # (260, 3, b'ab')
```

Методы применимые только к строкам

- Методы приведенные ниже ориентированы на специфические операции с символами Unicode и для байт смысла не имеют
- Примеры:

```
s1 = '123'
s1.casefold()     # более жесткий вариант метода lower()
s1.isdecimal()    # Unicode вариант метода isdigit()
s1.isidentifier() # является ли строка идентификатором ?
s1.isnumeric()    # является ли символы числами, например дробью ?
s1.isprintable()  # символ печатаемый (не пробел) ?

s1.format()       # форматирование переменных в строку
s1.format_map()   # форматирование словаря в строку
```

Форматирование строк

для печати

Использование оператора % с классом str

- Оператор % примененный к строке позволяет форматировать строку в стиле функции printf() языка C
- Некоторые спецификации формата:

```
'%d' - десятичное число
'%x' - шестнадцатеричное число
'%f' - число с плавающей запятой в виде 27.12
'%e' - число с плавающей запятой в виде 2.712e+01
'%s' - строка
'%6d' - десятичное число в поле из 6 символов
'%-6d' - то же с выравниванием влево
'%%' - символ %
'%p' - не поддерживается - не полное совпадение с языком C

Примеры:
s2 = '%3.1f' % 27.12
print(s2) # => 27.1
# Для нескольких значений круглые скобки справа от % обязательны
s = '%d %s %d года' % (9, 'сентября', 2019)
print(s) # => 9 сентября 2019 года
```

Метод format

- Метод format() позволяет форматировать строку не заботясь о типе используемых параметров
- В качестве спецификации формата используется конструкция {}, {номер_аргумента} или {имя_аргумента}
- Для точного форматирования после номера или имени аргумента может следовать двоеточие, после которого располагаются модификаторы формата
- Для выравнивания влево или вправо используются символы < и >

```
Примеры:
s = '{} {} {} года'.format(9, 'сентября', 2019)
print(s) # => 9 сентября 2019 года
s2 = '{2} {1} {0} года'.format(9, 'сентября', 2019)
print(s2) # => 2019 сентября 9 года
s3 = '{:3.1f}'.format(27.12)
print(s3) # => 27.1
```

Сборка строки оператором +

- Оператор + соединяет две строки в одну
- Почти любой объект можно преобразовать в строку встроенной функцией str()

```
Примеры:
s = str(9) + ' ' + 'сентября' + ' ' + str(2019) + ' года'
print(s) # => 9 сентября 2019 года
s2 = str(27.12)
print(s2) # => 27.12
```

Три способа форматирования строки, выводы:

- В отличие от Perl, PHP или bash в Питоне переменные внутри строк не интерпретируются
- В отличие от C и C++ форматирование происходит при создании текстовой строки, а не на уровне операций ввода-вывода

Операторы и их приоритеты

Таблица операторов
в порядке обратном приоритету их исполнения [1]
(таблица 5.2 на стр. 157 или на стр. 109 в оригинальном издании)

Операторы	Описание
yield x	Поддержка протокола send в функциях-генераторах
lambda args: expression	Создание анонимной функции

x if y else z	Трехместный оператор выбора (значение x вычисляется только если значение y истинно)
x or y	Логический оператор ИЛИ (значение y вычисляется только если значение x ложно)

x and y	Логический оператор И (значение y вычисляется только если значение x истинно)
not x	Логическое отрицание
x in y x not in y	Проверка вхождения объекта в коллекцию
x is y x is not y	Проверка идентичности объектов
x < y x <= y x > y x >= y	Операторы сравнения, проверка на подмножество и надмножество
x == y x != y	Операторы проверки на равенство

x | y	Битовая операция ИЛИ, объединение множеств
x ^ y	Битовая операция "Исключающее ИЛИ" (XOR), симметрическая разность множеств
x & y	Битовая операция И, пересечение множеств
x << y x >> y	Сдвиг значения x влево или вправо на y бит
x + y	Сложение, для последовательностей конкатенация
x - y	Вычитание, разность множеств
x * y	Умножение, для последовательностей повторение
x % y	Остаток от деления, форматирование строки в стиле printf()
x / y x // y	Деление истинное и деление с округлением вниз

-x +x	Унарный минус, унарный плюс
~x	Битовая операция НЕ (инверсия)
x ** y	Возведение в степень
x[i]	Индексация для последовательностей и словарей
x[i:j:k]	Извлечение фрагмента (slice) для последовательностей
x(...)	Вызов функции или другого callable объекта
x.attr	Обращение к атрибуту объекта
(...)	Кортеж, подвыражение, генератор кортежа
[...]	Список, генератор списка
{...}	Словарь, множество, генератор словаря или множества

Круглые скобки

- *Оператор выделения подвыражения* (круглые скобки) имеет один из самых высоких приоритетов
- Пример использование скобок для выделения подвыражений:
  - a = 7 + 3 * 10 # => 37
  - b = (7 + 3) * 10 # => 100
- Синтаксис использования круглых скобок:
  - вызов функции: перед открывающейся скобкой нет оператора, но есть выражение
  - литерал кортежа: это не вызов функции, но внутри скобок присутствуют запятые или внутри скобок пусто
  - генератор кортежа: внутри скобок присутствует ключевое слово *for*
  - круглые скобки это часть синтаксиса инструкции *def*
  - остальные случаи - оператор выделения подвыражения

Операции над объектами типов определенных пользователем

- Программист может определять классы, которые станут типами для создаваемых объектов (user-defined types)
- Стандартная библиотека языка Python это по большей части библиотека классов, фактически библиотечные типы это также user-defined types
- В классах могут быть определены реализации операторов. Объекты таких типов могут быть использованы в выражениях.
- Ортогональность языка программирования
  - применимость большинства операций к наибольшему числу типов объектов делает язык более ортогональным
  - ортогональный язык программирования компактнее и оттого проще в изучении
- Создавая собственные типы данных старайтесь сохранять ортогональность