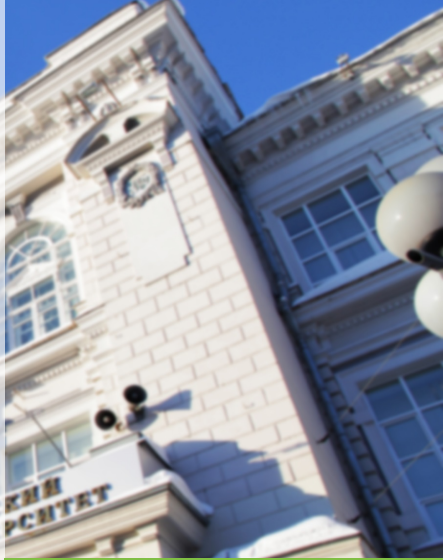
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

tpu.ru

Углубленный курс информатики

Лекция 3
Структуры данных:
строки, списки и кортежи



21 февраля 2022 г.

Долганов Игорь Михайлович,
к.т.н., доцент ОХИ ИШПР

ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Содержание

ТОМСКИЙ
ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ

Строки в Python

Основы строк

- Строки в Python относятся к *неизменяемым последовательностям*, что говорит о том, что содержащиеся в них символы имеют позиционный порядок слева направо и не могут быть изменены на месте.
- Строки в Python исполняют ту же роль, что и массивы символов в C-подобных языках, но в сравнении с массивами, они обладают инструментарием более высокого уровня.
- Строки в Python снабжены мощным набором инструментов обработки. Кроме того, в отличие от языков, подобных C, в Pyton не предусмотрен отдельный тип для индивидуальных символов, вместо этого применяются односимвольные строки.
- В плане обработки строки поддерживают операции *выражений*, такие как конкатенация (объединение строк), срезы (извлечение частей), индексация и т.д. Кроме операций выражений Python предоставляет набор *методов* строк, которые реализуют общие задачи, специфичные для строк.

Синтаксис строк

Python предлагает довольно обширный набор инструментов для работы со строками и в большинстве случаев использование строк проходит довольно легко. Вот несколько возможных способов записи строк в Python-коде:

- одинарные кавычки – `'spa"m'`;
- двойные кавычки – `"spa'm"`;
- тройные кавычки – `'''... spam ...'''`;
- неформатированные строки – `r'C:\new\test.spm'`;
- прочие способы записи строк.

Формы с одинарными и двойными кавычками безусловно являются наиболее распространенными; другие используются в особых случаях.

Строки в одинарных и двойных кавычках

Символы одинарных и двойных кавычек имеют одинаковое значение – обе формы записи работают абсолютно одинаково и возвращают объект одного типа:

```
>>> 'compounds', "compounds"
('compounds', 'compounds')
```

Можно внедрять символ одинарной кавычки в строку, заключенную в символы двойной кавычки, и наоборот:

```
>>> 'compound"s', "compound's"
('compound"s', "compound's")
```

При необходимости можно также внедрять символы кавычек, экранируя их при помощи обратной косой черты:

```
>>> 'knight\'s', "knight\"s"
("knight's", 'knight"s')
```

Многострочные блочные строки

- В Python имеется формат строковых литералов с тройными кавычками, иногда называемый *блочной строкой*.
- Такая форма начинается с трех кавычек (одинарных или двойных), за которыми следует произвольное количество строк текста, и заканчивается теми же самыми утроенными кавычками, что использовались в начале.
- Одинарные и двойные кавычки, встроенные в текст строки, могут отменяться, но не обязательно – строка не закончится, пока интерпретатор Python не встретит три неотмененных кавычки того же вида, который был использован в начале литерала.

```
>>> mantra = """Always look
... on the bright
... side of life."""
>>> mantra
'Always look\n on the bright\n side of life.'
```

Многострочные блочные строки

- Python собирает весь текст между тройными кавычками в единственную многострочную строку с внедренными символами новой строки (`\n`) в местах, где в коде присутствуют разрывы строк.
- Для того, чтобы посмотреть, как интерпретируется строка с символами новой строки, ее необходимо вывести с помощью функции `print`:

```
>>> print(mantra)
Always look
on the bright
side of life.
```

Операции над строками

1. Встроенная функция `len` возвращает длину строки:

```
>>> len("abc")
3
```

2. *Конкатенация* (сложение) строк выполняется при помощи операции `+` и создает новый объект строки с объединенным содержимым ее операндов:

```
>>> "abc" + "def"
"abcdef"
```

3. Повторение выполняется при помощи операции `*` и идентично добавлению строки к самой себе несколько раз:

```
>>> 'Hi!' * 4   #Повторение: то же, что 'Hi!'+ 'Hi!'+ ...
'Hi!Hi!Hi!Hi!'
```

Операции над строками

Например, можно вывести на экран формулу *н-декана*, используя операции конкатенации и повторения:

```
>>> 'CH3-' + 'CH2-' * 8 + 'CH3'
'CH3-CH2-CH2-CH2-CH2-CH2-CH2-CH2-CH2-CH3'
```

Для того, чтобы вывести формулу без кавычек, можно использовать встроенную функцию `print()`:

```
>>> nC10 = 'CH3-' + 'CH2-' * 8 + 'CH3'
>>> print(nC10)
CH3-CH2-CH2-CH2-CH2-CH2-CH2-CH2-CH2-CH3
```

Операции индексации

- Строки являются упорядоченными коллекциями символов и поэтому поддерживают доступ к своим элементам по индексу.
- *Индексация* – предоставление индекса желаемого компонента в квадратных скобках после имени, с которым связан объект строки. Результатом будет являться односимвольная строка в указанной позиции.
- Индексы в Python начинаются с 0 и заканчиваются величиной, на единицу меньше, чем длина строки.
- Python разрешает получать элементы из последовательностей с использованием *отрицательных* индексов.

Операции индексации


```
In [1]: s = "chemistry"

In [2]: s[1], s[-2]
Out[2]: ("h", "r")

In [3]: for i in range(len(s)):
   ...:     print(s[i].upper())
   ...:
C
H
E
M
I
S
T
R
Y
```

Операции срезов

- **Срезы** – обобщенная форма индексации для получения целого *сегмента* вместо одиночного элемента.
- При выполнении среза Python извлекает элементы, начиная с нижней границы и заканчивая, но не включая верхнюю границу, и возвращает новый объект, содержащий извлеченные элементы.
- Если левая и/или правая границы не указаны, по умолчанию для них принимаются индексы 0 и длина последовательности, соответственно.

```
In [1]: s = "chemistry"

In [2]: s[1:3], s[1:], s[:-1]
Out[2]: ("he", "hemistry", "chemistr")
```

Расширенные срезы

- В Python для выражений срезов есть поддержка опционального третьего индекса, используемого в качестве *шага*;
- Шаг прибавляется к индексу каждого извлеченного элемента.
- Полная форма среза выглядит следующим образом:

```
x[i:j:k]
```

что означает «извлечь элементы из `x`, начиная с индекса `i` и заканчивая индексом `j-1`, с шагом `k`»;

- Третий предел, `k`, по умолчанию, равен $+1$ и поэтому все элементы в срезе обычно извлекаются слева направо. Однако если указать явное значение, то можно применить третий предел для пропуска элементов или смены порядка их следования на противоположный.

Расширенные срезы

- Например, срез `x[1:10:2]` вернет *каждый второй элемент* из `x` в рамках индексов 1-9, т.е. элементы с индексами 1, 3, 5, 7 и 9.
- По аналогии, верхний и нижний пределы по умолчанию принимаются равными 0 и длине последовательности, соответственно, поэтому `x[::2]` вернет каждый второй элемент с начала и до конца последовательности:

```
In [1]: s = 'Beautifulisbetterthanugly'

In [2]: s[1:10:2]   #Пропуск элементов
Out[2]: 'euiui'

In [3] s[::2]
Out[3] 'Batflsetrhngy'
```

Расширенные срезы

- Можно также использовать отрицательный шаг для получения элементов в обратном порядке. Например, выражение среза `'spam'[::-1]` вернет новую строку `'maps'` – шаг -1 указывает, что срез должен идти справа налево, а не слева направо:

```
In [4] s = 'spam'

In [5] s[::-1]   #Смена порядка элементов на противоположный
Out[5] 'maps'
```

- При отрицательном шаге смысл нижней и верхней границ по сути меняется на противоположный. Таким образом, срез `x[5:1:-1]` получает элементы со второго по пятый в обратном порядке (элементы с индексами 5, 4, 3 и 2):

```
In [6]  s = 'Simpleisbetterthancomplex'

In [7] s[5:1:-1]   #Смысл границ изменяется
Out[7] 'elpm'
```

Методы строк

В таблице приведена сводка мо методам строк; они часто меняются, поэтому используйте руководство по стандартной библиотеке Python для получения самого актуального списка или используйте функции `dir` или `help` для любой строки (либо имени типа `str`). В таблице имя `s` – это объект строки, а опциональные (необязательные) аргументы указаны в квадратных скобках.

`s.capitalize()`	`s.isdigit()`
`s.casefold()`	`s.lower()`
`s.center(width[, fillchar])`	`s.islower()`
`s.count(sub[, start[, end]])`	`s.isnumeric()`
`s.endswith(suffix[, start[, end]])`	`s.istitle()`
`s.find(sub[, start[, end]])`	`s.removeprefix(prefix, /)`
`s.format(*args, **kwargs)`	`s.removesuffix(suffix, /)`
`s.format_map(mapping)`	`s.replace(old, new[, count])`
`s.index(sub[, start[, end]])`	`s.isupper()`
`s.isalnum()`	`s.split(sep=None, maxsplit=-1)`
`s.isalpha()`	`s.startswith(prefix[, start[, end]])`
`s.isascii()`	`s.join(iterable)`
`s.isdecimal()`	`s.upper()`

Использование методов строк

- Если нужно заменить подстроку, то можно применить метод `replace`:

```
In [1]: s = "CH3-CH2-CH2-CH3"

In [2]: s.replace("CH2-", "CHOH-")
Out[2]: 'CH3-CHOH-CHOH-CH3'

```

- Метод `find` возвращает индекс, по которому будет найдена подстрока (по умолчанию поиск начинается сначала исходной строки) или -1, если подстрока не будет найдена:

```
In [3]: s = "CH3-CH2-CH2-CH3"

In [4]: s.find("CH2")
Out[4]: 4

```

Метод format

- Метод format использует в качестве шаблона строку, для которой вызывается и принимает произвольное количество аргументов, которые являются значениями для подстановки в шаблон.

```
>>> template = '0, 1 and 2'

>>> template.format('pizza', 'burger', 'buritto')
'pizza, burger and buritto'

>>> template = 'Italy, USA and UMS'

>>> template.format(Italy='pizza', USA='burger', UMS='buritto')
'pizza, burger and buritto'

>>> template = 'Italy, 0 and UMS'

>>> template.format('burger', Italy='pizza', UMS='buritto')
'pizza, burger and buritto'

>>> template = '{}, {} and {}'

>>> template.format('pizza', 'burger', 'buritto')
'pizza, burger and buritto'
```

Примеры использования метода format


```
>>> '{0:10} = {1:10}'.format('number', 123.5476)
'number     =   123.5476'

>>> '{0:>10} = {1:<10}'.format('number', 123.5476)
'     number = 123.5476  '
```

{0:10} означает первый позиционный аргумент в поле шириной 10 символов, {1:<10} – второй позиционный аргумент, выровненный влево в поле шириной 10 символов.

Во всех случаях номер аргумента можно не указывать, если аргументы выбираются слева направо при помощи автоматической нумерации, хотя это и делает код менее читаемым:

```
>>> '{:10} = {:10}'.format('number', 123.5476)
'number     =   123.5476'

>>> '{:>10} = {:<10}'.format('number', 123.5476)
'     number = 123.5476  '
```

Примеры использования метода format

Для чисел с плавающей точкой определены специальные коды форматирования. Например, в приведенных ниже инструкциях, `{2:g}` означает, что третий аргумент форматируется по умолчанию в соответствии с представлением чисел с плавающей точкой `g`, `{1:2f}` указывает формат с плавающей точкой `f` с двумя знаками после запятой, а `{2:06.2f}` добавляет поле шириной 6 символов и дополнением нулями слева:

```
>>> '{0:e}, {1:.3e}, {2:g}'.format(3.14519, 3.14519, 3.14519)
'3.145190e+00, 3.145e+00, 3.14519'

>>> '{0:f}, {1:.2f}, {2:06.2f}'.format(3.14519, 3.14519, 3.14519)
'3.145190, 3.15, 003.15'
```

Форматированные строки (f-строки)

- Литерал форматированных строк или f-строки – это строковый литерал с префиксом `f` или `F`. Данные строки могут содержать замещающие поля, которые являются выражениями в фигурных скобках .
- Мини-язык для спецификатора формата такой же, как и в методе `format`.

```
In [1]: f"{1.2354:.2f}"
Out[1]: '1.24'

In [2]: f"{1.2354:e}"
Out[2]: '1.235400e+00'

In [3]: f"{1.2354:.3e}"
Out[3]: '1.235e+00'

In [4]: f"{1.2354:g}"
Out[4]: '1.2354'
```

ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Списки

Списки

- Списки в Python – наиболее гибкая разновидность объектов упорядоченных коллекций.
- Списки могут содержать объекты любого типа: строки, числа или другие списки.
- Списки *можно изменять* на месте присваиванием по индексам или с использованием срезов, вызвав специальные методы или выполнив оператор удаления и т.д.

Операция	**Описание**
`a = []`	Пустой список
`a = [123, 'abc', 1.354, []]`	Четыре элемента: индексы 0...3
`a = ['Joe', 30.0, ['dev', 'prof']]`	Вложенные списки
`a = list('hello')`	Список элементов итерируемого объекта
`a = list(range(-5, 6))`	Список последовательных целых чисел
`a[i]`	Индекс
`a[i][j]`	Индекс индекса
`a[i:j]`	Срез
`len(a)`	Длина
`a1 + a2`	Конкатенация
`a * 3`	Повторение
`x in a`	Вхождение
`a.append(5)`	Добавление элемента в конец списка
`a.extend([10, 20, 30])`	Добавление списка в конец исходного списка

Базовые операции со списками

- Списки являются *последовательностями*, поэтому поддерживают многие операции, характерные для строк.
- Например, для списков определены операторы + и *. Данные операторы, также как и в случае со строками, означают конкатенацию и повторение, только возвращают в качестве результата новый список, а не строку.

```
>>> len([1, 2, 3, 4, 5])                    #Длина
5

>>> [1, 2, 3, 4, 5] + [6, 7, 8, 9, 10]      #Конкатенация
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> ["Hi!"] * 5                             #Повторение
["Hi!", "Hi!", "Hi!", "Hi!", "Hi!"]
```

Итерация по спискам

В общем смысле для списков определены все операции над последовательностями, в том числе и инструменты итерации:

```
>>> "banana" in ["banana", "orange", "apple"] #Проверка вхождения
True

>>> for fruit in ["banana", "orange", "apple"]: #Итерация
...     print(fruit, end=" ")
...
banana orange apple

```

Оператор цикла `for` проходит (итерируется) по всем элементам в любой последовательности (итерируемом объекте) слева направо, выполняя операторы для каждого из них.

Генераторы списков (list comprehension)

Генераторы списков – это способ создания нового списка с применением выражения к каждому элементу последовательности (по факту в любом итерируемом объекте).

```
>>> res = [c * 4 for c in "HELLO"]

>>> res
["HHHH", "EEEE", "LLLL", "LLLL", "OOOO"]

```

- Генераторы списков записываются более кратко и выполняются чуть быстрее.
- В некоторых сложных случаях стоит отдать предпочтение использованию цикла `for` из-за его более высокой читаемости.

```
>>> res = []

>>> for c in "HELLO":
...     res.append(c * 4)
...

>>> res
["HHHH", "EEEE", "LLLL", "LLLL", "OOOO"]

```

Индексация и срезы

- Индексация и срезы для списков работают аналогично тому, как это было описано для объектов строк.
- Результатом индексации списка может быть объект любого типа, находящийся по указанному индексу, тогда как срезы всегда возвращают новый объект списка.

```
>>> fruits = ["banana", "orange", "apple"]

>>> fruits[2]              #Индексы начинаются с нуля
"apple"

>>> fruits[-2]             #Отрицательные индексы отсчитываются справа
"orange"

>>> fruits[1:3]            #Срезы получают сегменты
["orange", "apple"]

>>> fruits[-1]             #Результат среза всегда новый список
["apple"]
```

Вложенность списков

- Внутри списков могут содержаться вложенные списки или объекты других типов.
- Матрицы в Python можно представить в виде вложенных списков.
  Пример двумерного массива 3×3, построенный в виде списка:

```
>>> matrix = [[10, 20, 30], [40, 50, 60], [70, 80, 90]]
```

- Если указать один индекс, то будет получена целая строка, а при указании двух индексов будет возвращен элемент строки:

```
>>> matrix[1]
[40, 50, 60]

>>> matrix[2][0]
70

>>> matrix = [[10, 20, 30],
...           [40, 50, 60],
...           [70, 80, 90]]

>>> matrix[1][1]
50
```

Изменение списков

- Так как списки – *изменяемый* тип объектов, для них определены операции, которые могут модифицировать объект списка *на месте*.
- Операции модифицирования списков изменяют объект списка напрямую, перезаписывая его старое значение, без необходимости создания новой копии, как в случае работы со строками.

Присваивание по индексам и срезам

Содержимое списка может быть изменено присваиванием значения либо отдельному элементу (по его индексу), либо целому сегменту (по срезу):

```
>>> food = ["burger", "pizza", "buritto"]

>>> food[1] = "toast"  #Присваивание по индексу

>>> food
["burger", "toast", "buritto"]

>>> food[:2] = ["need", "more"]  #Присваивание срезу

>>> food
["need", "more", "buritto"]
```

Изменение списков

Примеры присваиваний по срезу

```
>>> a = [10, 20, 30]

>>> a[1:2] = [4, 5]       #Замена/вставка

>>> a
[10, 4, 5, 30]

>>> a[1:1] = [60, 70]     #Вставка (ничего не заменяется)

>>> a
[10, 60, 70, 4, 5, 30]

>>> a[1:2] = []           #Удаление (ничего не добавляется)

>>> a
[10, 70, 4, 5, 30]

>>> a[:0] = [1, 2, 3]       #Вставка на место :0, пустой срез в начале

>>> a
[1, 2, 3, 10, 70, 4, 5, 30]

>>> a[len(a):] = [4, 5, 6] #Вставка на место len(a):, пустой срез в конце

>>> a
[1, 2, 3, 10, 70, 4, 5, 30, 4, 5, 6]
```

Вызовы методов списков

- Подобно строкам, списки имеют набор специфичных методов, многие из которых ведут к изменению исходного списка на месте:

```
>>> a = ["eat", "more", "SPAM"]

>>> a.append("please")

>>> a
["eat", "more", "SPAM", "please"]

>>> a.sort()  #Сортировка элементов списка ("S" < "e")

>>> a
["SPAM", "eat", "more", "please"]

```

- Наиболее распространенный метод – `append` добавляет объект в конец списка.
- Эффект выполнения выражения `a.append(x)` аналогичен `a + [x]` с одним принципиальным отличием: первый вариант изменяет `a` на месте, а второй вариант создает новый объект списка.
- Метод `sort` упорядочивает элементы в списке.

Вызовы методов списков

- Метод `reverse` изменяет порядок элементов в списке на противоположный (обращает список) на месте.
- Метод `extend` добавляет множество элементов в конец списка.
- Метод `pop` удаляет и возвращает последний элемент списка, если не указана конкретная позиция.

```
>>> a = [0, 1]

>>> a.extend([2, 3, 4])  #Добавление множества элементов в конец списка

>>> a
[0, 1, 2, 3, 4]

>>> a.pop()              #Удаление и возврат последнего элемента
4

>>> a
[0, 1, 2, 3]

>>> a.reverse()          #Метод обращения списка на месте

>>> a
[3, 2, 1, 0]
```

Вызовы методов списков

- `remove` – удаление элементов списка;
- `insert` – вставка элементов по индексу;
- `count` – подсчет количества вхождений;
- `index` – нахождение индекса элемента.

```
>>> a = ["spam", "eggs", "ham"]
>>> a.index("spam")                    #Поиск индекса объекта
0

>>> a.insert(1, "toast")               #Вставка по индексу
>>> a
["spam", "toast", "eggs", "ham"]

>>> a.remove("spam")                   #Удаление по значению
>>> a
["toast", "eggs", "ham"]

>>> a.pop(1)                           #Удаление по индексу
"eggs"

>>> a
["toast", "ham"]

>>> a.count("ham")                     #Количество вхождений
1
```

Дополнительные сведения о методе sort

- В методе sort аргумент reverse позволяет производить сортировку в порядке убывания вместо возрастания, а параметр key задает функцию с одним аргументом, которая возвращает значение для использования при сортировке.

```
>>> a = ["abc", "ABD", "aBe"]

>>> a.sort()                                #Сортировка со смешанным регистром
>>> a
["ABD", "aBe", "abc"]

>>> a = ["abc", "ABD", "aBe"]

>>> a.sort(key=str.lower)                   #Приведение к нижнему регистру
>>> a
["abc", "ABD", "aBe"]

>>> a.sort(key=str.lower, reverse=True) #Изменение порядка сортировки

>>> a
["aBe", "ABD", "abc"]
```

ТОМСКИЙ
ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ

Кортежи

Кортежи

- Кортежи служат для хранения нескольких объектов вместе.
- Аналог списков, но без обширной функциональности *класса списков*.
- Кортежи *неизменяемы*, так же, как и строки.

Операция	Описание
`()`	Пустой кортеж
`t = (0, )`	Одноэлементный кортеж (не выражение)
`t = (0, 'Hi', 1.2, 3)`	Кортеж из четырех элементов
`t = 0, 'Hi', 1.2, 3`	Такой же кортеж, как в предыдущей строке
`t = ('John', ('prof', 'dev'))`	Вложенные кортежи
`t = tuple('hello')`	Кортеж из элементов итерируемого объекта
`t[i]`	Индекс
`t[i][j]`	Индекс индекса
`t[i:j]`	Срез
`len(t)`	Длина кортежа
`t1 + t2`	Конкатенация
`t * 3`	Повторение
`'spam' in t`	Проверка вхождения
`t.index('Hi')`	Поиск индекса элемента
`t.count('hello')`	Подсчет повторений элемента

Операции над кортежами

- Кортежи поддерживают обычные операции, специфичные для последовательностей:

```
>>> (10, 20) + (30, 40)          #Конкатенация
(10, 20, 30, 40)

>>> (1, 2) * 5
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)   #Повторение

>>> t = (10, 20, 30, 40, 50)

>>> t[0], t[1:3]                 #Индексация, срезы
(10, (20, 30))
```

- Создание кортежа из одного элемента:

```
>>> x = (30)      #Целое число

>>> x
30

>>> y = (30, )    #Кортеж, содержащий целое число

>>> y
(30,)
```

Сортировка кортежей

Чтобы отсортировать элементы кортежа, потребуется сначала преобразовать его в список, для получения доступа к методу сортировки:

```
>>> t = ("b", "c", "a", "e", "d")

>>> tmp = list(t)            #Создание списка из элементов кортежа

>>> tmp.sort()               #Сортировка списка

>>> tmp
["a", "b", "c", "d", "e"]

>>> t = tuple(tmp)           #Создание кортежа из элементов списка

>>> t
("a", "b", "c", "d", "e")

```

Встроенные функции `list` и `tuple` используются для преобразования объекта в список, а затем в кортеж; на самом деле оба вызова создают новые объекты, но внешне это похоже на преобразование.

Преобразование кортежей

Преобразовать элементы кортежа можно при помощи выражения генератора списка, либо при помощи итерации по его элементам в цикле `for`:

С использованием генератора списка:

```
>>> t = (1, 2, 3, 4, 5)

>>> a = [x ** 3 for x in t]

>>> a
[1, 8, 27, 64, 125]

>>> t = tuple(a)
>>> t
(1, 8, 27, 64, 125)

```

С использованием цикла `for`:

```
>>> t = (1, 2, 3, 4, 5)

>>> a = []

>>> for element in t:
...     a.append(element ** 3)

>>> t = tuple(a)
>>> t
(1, 8, 27, 64, 125)

```

Преобразование кортежей

Изменять элементы кортежей по индексу, как и элементы строк, нельзя:

```
>>> t = (1, 2, 3, 4, 5)

>>> t[1] = "hi!"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: "tuple" object does not support item assignment

>>> s = "compound"

>>> s[3] = "b"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: "str" object does not support item assignment

```

ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Контакты

Игорь Михайлович Долганов,
к.т.н., доцент ОХИ ИШПР

Учебный корпус №2, ауд. 136
dolganovim@tpu.ru
+7-960-978-43-07

Благодарю за внимание!