

Структуры данных и функции

1. Изменяемые и неизменяемые типы данных

Строки

Списки

Кортежи (tuple vs named tuple)

Сеты (+frozen set)

Словари

Файлы, режимы чтения (+ контекстные менеджеры)

1. Функции

Аргументы

Namespaces

Scopes (LEGB)

Enclosing

Function as an object

1. Функциональное программирование

Парадигмы программирования

Динамическая типизация и какие еще лейблы можно наклеить на Python

List comprehensions

Lambda, map, filter, reduce, zip

functools, itertools - basics

Декораторы

Области видимости

Структура данных

программная единица, к функциям которой относят хранение и обработку множества однотипных или логически связанных данных в вычислительной технике, реализованную через интерфейс, состоящий из набора функций для добавления, поиска, изменения и удаления данных.

50 3 оттенка термина “структура данных”:

- Абстрактный тип данных;
- Реализация какого-либо абстрактного типа данных;
- Экземпляр типа данных (конкретный список);

Структура данных = типы данных + ссылки + операций над ними в выбранном языке программирования.

- Тип данных - множество значений и операций на этих значениях.
- Ссылка - это объект, указывающий на определенные данные, но не хранящий их.

В начале был PEP8

Предпосылки:

Код читают чаще, чем пишут. Давайте работать над его читаемостью.

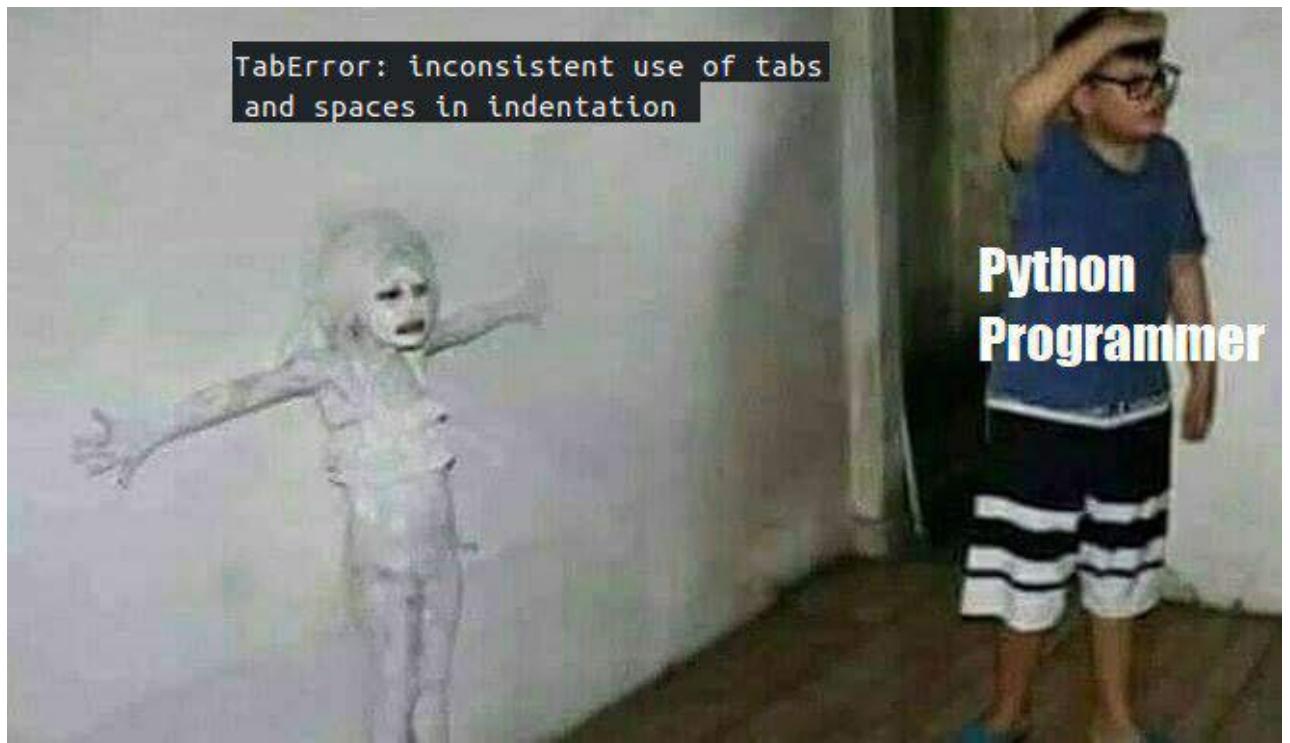
Поработали:

<https://www.python.org/dev/peps/pep-0008/> (<https://www.python.org/dev/peps/pep-0008/>)

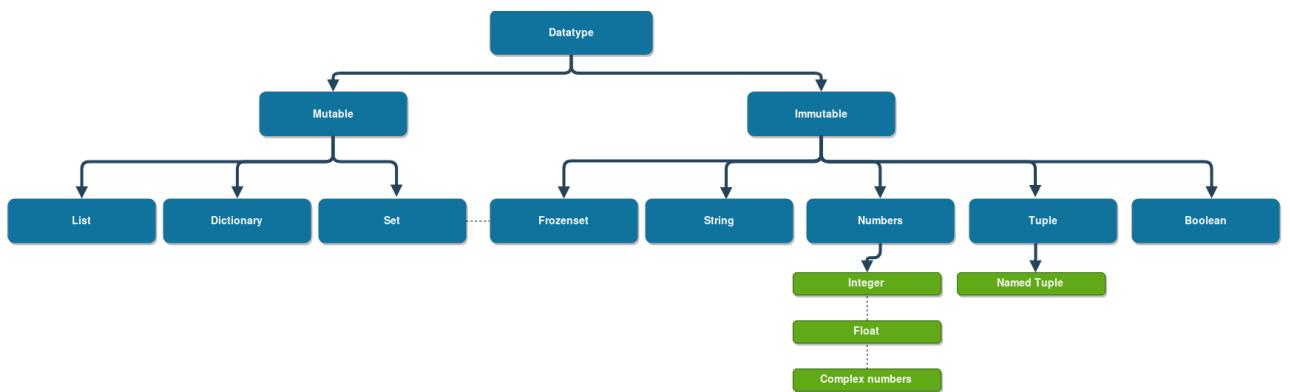
Пример того, за что можно получить по рукам

```
# Yes:  
import os  
import sys  
from subprocess import Popen, PIPE  
  
# No:  
import sys, os  
from sklearn import *  
  
"""Limit all lines to a maximum of 79 characters!"""  
===== looong = "oooooooooooooooooooooooooooooooooooo...o"
```

PEP 8: line too long (98 > 79 characters)



Как настроить своего питона - <https://www.jetbrains.com/help/pycharm/code-inspection.html> (<https://www.jetbrains.com/help/pycharm/code-inspection.html>)



Строки

это упорядоченные последовательности символов, используемые для хранения и представления текстовой информации.

Литералы строк

```
In [267]: "Апостроф или кавычка?"  
'Апостроф или кавычка?'  
"""Апостроф или три кавычки"""  
"Апостроф"" или кавычка?"  
"Апостроф " " или кавычка?"
```

```
Out[267]: 'Апостроф или кавычка?'
```

Записываем многострочные блоки текста

```
In [158]: big = '''a very very  
... very big  
... string'''  
  
print(big)
```

```
a very very  
very big  
string
```

Экранирование

это замена в тексте управляемых символов на соответствующие текстовые подстановки.

```
In [159]: problem = 'C:\\teeeeeext.txt'  
no_problem = r'C:\\teeeeeext.txt'  
no_problem_2 = 'C:\\\\teeeeeext.txt'  
  
print(problem, "\\n", no_problem, "\\n", no_problem_2)
```

```
C:      eeeeeext.txt  
C:\\teeeeeext.txt  
C:\\teeeeeext.txt
```

Экранированная последовательность	Назначение
\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)

Проблема:

Иногда мы хотим избавить backslash в тексте от рабских функций. Но в питоне так не принято...

Raw string - освободитель backslash'ей!



Формально:

Если перед открывающей кавычкой стоит символ 'r' (в любом регистре), то механизм экранирования отключается.

БОЛЬ Кодировки

ASCII (American Standard Code for Information Interchange) - стандарт кодирования символов, а именно: латинского алфавита, цифр и некоторых символов.

Unicode - стандарт кодирования символов, включающий в себя знаки почти всех письменных языков мира. В настоящее время стандарт является доминирующим в Интернете.

```
In [160]: print(f"Функция ord() unicode-символ переводит в его код: {2} - ", ord("2"),
      f"\nФункция chr() код ASCII переводит в символ: {50} - ", chr(50))
```

Функция `ord()` символ переводит в его код ASCII: 2 - 50
Функция `chr()` код ASCII переводит в символ: 50 - 2

Разница между ascii(), str(), repr()

Общее

Методы для получения строкового представления объектов.

Кратко об отличиях

`repr` - однозначный (“official” string representation of an object)

`str` - человекочитаемый (nicely printable string representation of an object)

`ascii` - родственник `repr`'а, заменяющий не-ASCII символы на escape characters

Прямо в тексте

%s-спецификатор - конвертирует, используя `str()`

%r - конвертирует, используя `repr()`

%a - конвертирует, используя `ascii()`

```
In [268]: import datetime
today = datetime.datetime.now()

print("str() -", str(today),
      "\nrepr() -", repr(today))

mot = "Les garçons"
print("\nrepr() -", repr(mot),
      "\nascii() -", ascii(mot))
```

```
str() - 2019-05-21 19:01:15.710211
repr() - datetime.datetime(2019, 5, 21, 19, 1, 15, 710211)

repr() - 'Les garçons'
ascii() - 'Les garçons'
```

Метод - это функция, которая применяется к объекту определенного типа и позволяет делать какие-либо манипуляции с ним.

```
In [162]: example = "Python is awesome"  
example
```

```
Out[162]: 'Python is awesome'
```

Метод `find()` - позволяет найти подстроку в строке и вывести ее индекс.

```
In [163]: example.find("aw")
```

```
Out[163]: 10
```

А если подстроки в строке нет?

При отсутствии подстроки в строке выводит -1.

```
In [164]: example.find("l")
```

```
Out[164]: -1
```

Ищем с правого конца

возвращает номер последнего вхождения подстроки

```
In [165]: example.rfind("aw")
```

```
Out[165]: 10
```

А как еще найти индекс подстроки?

```
In [166]: example.index("Py")
```

```
Out[166]: 0
```

И в чем тогда разница между index и find?

```
In [167]: example.index("Snake")
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-167-accb9d7cb2ae> in <module>  
----> 1 example.index("Snake")
```

```
ValueError: substring not found
```

Упс.. Оказывается, индекс умеет ругаться :(

Строки можно конкатенировать

```
In [168]: first = "lenin"  
second = "grib"  
third = first + " " + second + " "  
print(third)  
  
# И склеивать по заданному символу  
" ".join([first, second])
```

```
lenin grib
```

```
Out[168]: 'lenin grib'
```

И дублировать

```
In [169]: print(third*10, "\n")
```

```
lenin grib lenin grib lenin grib lenin grib lenin grib lenin grib len  
in grib lenin grib lenin grib
```

Строка как торт - слайсинг



```
In [170]: print(f"Слайсим:{repr(third)}\n"+third[:5], third[-5:-1], third[::-2]) # шагаем  
# А еще мы можем разрезать по швам  
print("Применяем метод .split():", third.split(" "))
```

```
Слайсим:'lenin grib '  
lenin grib  ignnl  
Применяем метод .split(): ['lenin', 'grib', '']
```

Хотим изменений

```
In [171]: third.replace("grib ", "molodec!")
```

```
Out[171]: 'lenin molodec!'
```

Балуемся кортежами

```
In [172]: third.partition("nin")
```

```
Out[172]: ('le', 'nin', ' grib ')
```

Когда размер имеет значение

```
In [173]: ecclesiastes = "vanitas vanitatum et omnia vanitas";
          print(ecclesiastes.zfill(46))
```

```
000000000000vanitas vanitatum et omnia vanitas
```

```
In [174]: ecclesiastes.center(46, " ")
```

```
Out[174]: ' vanitas vanitatum et omnia vanitas '
```

```
In [175]: print(".ljust(): ", ecclesiastes.ljust(46, "E"),
          "\n.rjust(): ", ecclesiastes.rjust(46, "E"))
```

```
.ljust(): vanitas vanitatum et omnia vanitasEEEEEEEEEEEEE
.rjust(): EEEEEEEEEEvanitas vanitatum et omnia vanitas
```

Или так

S.lstrip([chars]) - удаляем пробельные символы в начале строки

S.rstrip([chars]) - удаляем пробельные символы в конце строки

S.strip([chars]) -удаляем пробельные символы в начале и в конце строки

~~Спойлер к дезе~~

Возвращаем количество непересекающихся вхождений подстроки в диапазоне [начало, конец]

```
In [176]: "ahahaahauhuhhahahahahehehehe".count("ah")
```

```
Out[176]: 8
```

Играем с регистрами

```
In [177]: print("Upper case - ", third.upper(), "\n",
      "Title - ", third.title(), "\n",
      "Lower case - ", third.lower(), "\n",
      "Capitalize - ", third.capitalize(), "\n"
      "Swapcase - ", third.swapcase())
```

```
Upper case - LENIN GRIB
Title - Lenin Grib
Lower case - lenin grib
Capitalize - Lenin grib
Swapcase -  LENIN GRIB
```

```
In [178]: def capitalize_name(name):
           return name.capitalize()

name = "alice"
f"{capitalize_name(name)} is my friend"
```

```
Out[178]: 'Alice is my friend'
```

Ставим диагноз строкам

S.isdigit() - состоит ли строка из цифр

S.isalpha() - состоит ли строка из букв

S.isalnum() - состоит ли строка из цифр или букв

S.islower() - состоит ли строка из символов в нижнем регистре

S.isupper() - состоит ли строка из символов в верхнем регистре

S.istitle() Начинаются ли слова в строке с заглавной буквы

`Sisspace()` - состоит ли строка из неотображаемых символов (пробел, символ перевода страницы ('\f'), "новая строка" ('\n'), "перевод каретки" ('\r'), "горизонтальная табуляция" ('\t') и "вертикальная табуляция" ('\v'))

`S.startswith(паттерн)` - начинается ли строка S с паттерна

`S.endswith(паттерн)` - заканчивается ли строка S паттерном

Так почему же immutable..

```
In [179]: second[2] = "ë"
```

```
-----  
TypeError Traceback (most recent call last)  
<ipython-input-179-ec91b4022519> in <module>  
----> 1 second[2] = "ë"  
  
TypeError: 'str' object does not support item assignment
```

Остается только создавать новые объекты каждый раз, когда мы захотим модифицировать нашу строку

```
In [180]: replace_result = second.replace("i", "ë")  
print(id(replace_result), id(second))  
print(second, replace_result)
```

```
139803673335304 139803676806928  
grib grëb
```

Форматирование строк

Способ 1, для дедов - %

- становится нечитаемым на больших текстах
- легко может стать причиной возникновения ошибок

```
In [181]: price = "cheap"
name = "Linus Torvalds"
"'Talk is %s. Show me the code.' – %s" % (price, name)
```

```
Out[181]: "'Talk is cheap. Show me the code.' – Linus Torvalds"
```

Способ 2 - .format()

- появился в Python 2.6
- более здоровая альтернатива %-форматированию
- все равно вербозный способ

```
In [182]: "'Talk is {}. Show me the code.' – {}".format(price, name)
```

```
Out[182]: "'Talk is cheap. Show me the code.' – Linus Torvalds"
```

```
In [183]: "'Talk is {1}. Show me the code.' – {}".format(name, price)
```

```
Out[183]: "'Talk is cheap. Show me the code.' – Linus Torvalds"
```

```
In [184]: data = {"price": "cheap", "name": "Linus Torvalds"}  
         "'Talk is {price}. Show me the code.' – {name}".format(**data)
```

```
Out[184]: "'Talk is cheap. Show me the code.' – Linus Torvalds"
```

Способ 3 - just one simple.. great magnificent f

- царствование с Python 3.6 версии
- самый быстрый способ форматирования
- подробности в PEP 498



```
In [185]: f"'Talk is {price}. Show me the code.' – {name}"
```

```
Out[185]: "'Talk is cheap. Show me the code.' – Linus Torvalds"
```

Включаем критическое мышление

In [186]:

```
import timeit
pc_way = timeit.timeit("""price = "cheap"
name = "Linus Torvalds"
'Talk is %s. Show me the code. - %s' % (price, name)""", number=100000)
format_way = timeit.timeit("""price = "cheap"
name = "Linus Torvalds"
'Talk is {}. Show me the code. - {}'.format(price, name) """, number=100000)
f_way = timeit.timeit("""price = "cheap"
name = "Linus Torvalds"
f'Talk is {price}. Show me the code. - {name}'''", number=100000)
print("%-форматирование:", pc_way, "\n",
      "Метод .format()", format_way, "\n",
      "f-форматирование:", f_way, "\n")
```

%-форматирование: 0.03435833499679575
Метод .format() 0.04441355900053168
f-форматирование: 0.011435857995820697

А еще... Даже строка умеет в математику

In [187]: `f"{2 * 46}"`

Out[187]: '92'

Итак, мы вспомнили узнали

- Определение строк и методов
- Методы строк
 - Отличия index() и find()
 - Конкатенация и дублирование строк
 - Слайсинг
 - Методы смены регистров
- Форматирование строк
 - %-форматирование
 - Метод .format()
 - Божественный f



Список - упорядоченная изменяемая коллекция объектов произвольных типов.

Динамический массив ссылок.

Инициализировать пустой список можно через конструктор или через литералы

In [188]:

```
# через литералы
a = []
# через конструктор
b = list()
```

Список может содержать любое количество любых объектов (в т.ч. и вложенные списки) или не содержать ничего

```
In [189]: snake = list('hat')
noble_programmer = ["Knuth", "Tanenbaum", "bike"]
matrix = [[1, 2], [3, 4]]
hungry_list = []
```

Важно: при создании списка с начальным значением не происходит копирования этого значения

```
In [190]: matrix = [[0]] * 3  
matrix
```

```
Out[190]: [[0], [0], [0]]
```

```
In [191]: matrix[0][0] = 'Null'  
matrix
```

```
Out[191]: [['Null'], ['Null'], ['Null']]
```

Как правильно?

Основные методы list

Добавление элементов

- `append` и `extend` - добавляют в конец списка один элемент или некоторую последовательность.

```
In [192]: l = [1, 2, 3]
l.append(4)
l.extend((5, 6))
l
```

```
Out[192]: [1, 2, 3, 4, 5, 6]
```

- вставить элемент перед элементом с указанным индексом можно с помощью метода `insert`

```
In [193]: l = [1, 2, 3]
l.insert(0, 'Null')
print(l)
l.insert(-2, 'some')
print(l)
```

```
['Null', 1, 2, 3]
['Null', 1, 'some', 2, 3]
```

Замена

- можно заменить целую последовательность на другую:

In [194]:

```
l = [1, 2, 3]
l[:2] = [0] * 2
l
```

Out[194]:

```
[0, 0, 3]
```

Конкатенация списков

- результат конкатенации списка - всегда новый список.

In [195]:

```
l1 = [1, 2]
l2 = [3]
print(id(l1), id(l2))
print(id(l1 + l2))
```

```
139803672921352 139803672915400
139803672920968
```

Но есть возможность **inplace** конкатенации

In [196]:

```
l1 = [1, 2]
l2 = [3]
print(id(l1), id(l2))
l1 += l2
print(id(l1))
```

```
139803672922696 139803673082504
139803672922696
```

Удаление элемента

- удалить элемент или целую последовательность из списка можно с помощью оператора `del`, указав индекс.

```
In [197]: l = [1, 2, 3]
del l[:2]
l
```

```
Out[197]: [3]
```

```
In [198]: l = [1, 2, 3]
del l[:]
l
```

```
Out[198]: []
```

- с помощью метода `pop` можно получить в качестве возвращаемого значения удаляемый элемент

```
In [199]: l = [1, 2, 3]
first = l.pop(0)
print(first, l)
```

```
1 [2, 3]
```

- удалить первое вхождение элемента в список

```
In [200]: kebab = [1, 1, 3]
kebab.remove(1)
kebab
```

```
Out[200]: [1, 3]
```

Несколько других полезных методов

`list.index(x, [start [, end]])`

Возвращает положение первого элемента со значением x (при этом поиск ведется от start до end)

```
In [201]: l = [1, 2, 3]
l.index(1)
```

```
Out[201]: 0
```

Мальчик: оборачивает все в try;

Мужчина: знает, что и так все будет хорошо

```
In [202]: l = [1, 2, 3]
l.index(4)
```

```
-----  
ValueError                                Traceback (most recent call last)
<ipython-input-202-c2f7f05d3b2b> in <module>
      1 l = [1, 2, 3]
----> 2 l.index(4)
```

```
ValueError: 4 is not in list
```


`list.count(x)` Возвращает количество элементов со значением x

```
In [203]: l = [1, 1, 1, 2, 3]
l.count(1)
```

```
Out[203]: 3
```

```
In [204]: l = [1, 1, 1, 2, 3]
l.count(4)
```

```
Out[204]: 0
```

Как перевернуть список?

```
In [269]: l = [1, 2, 3]
a = l.reverse()
print(a)
l
```

None

Out[269]: [3, 2, 1]

```
In [206]: l = [1, 2, 3]
lr = l[::-1]
lr
```

Out[206]: [3, 2, 1]

```
In [207]: l = [1, 2, 3]
lr = reversed(l)
lr
```

Out[207]: <list_reverseiterator at 0x7f269441ac18>

Как сортировать список ?

```
In [208]: l = [3, 1, 2]
l.sort()
l
```

```
Out[208]: [1, 2, 3]
```

```
In [209]: l = [3, 1, 2]
ls = sorted(l)
ls
```

```
Out[209]: [1, 2, 3]
```

Функции `sorted` и методу `sort` можно опционально указать направление сортировки, а также функцию-ключ

```
In [210]: l = [3, 4, 2, 1]
l.sort(key=lambda x: x % 2, reverse=True)
l
```

```
Out[210]: [3, 1, 4, 2]
```

Сортировка и сравнение в Python 3.x не конвертирует типы, в отличии от Python 2.x

```
In [211]: a = [2, 1, '1']
a.sort()
```

```
-----  
TypeError                                 Traceback (most recent call last)
<ipython-input-211-3c3144ceca8d> in <module>
      1 a = [2, 1, '1']
----> 2 a.sort()  
  
TypeError: '<' not supported between instances of 'str' and 'int'
```

Генераторы списков aka list comprehensions

Специальная синтаксическая конструкция, которая позволяет по определенным правилам создавать заполненные списки. Их удобство заключается в более компактной записи программного кода в сравнении с созданием списка обычным способом.

Генератор списков - способ построить новый список, применяя выражение к каждому элементу последовательности. Генераторы списков очень похожи на цикл `for`.

```
In [212]: a = []
for i in range(10):
    a.append(i)

a_gen = [i for i in range(10)]
a == a_gen
```

```
Out[212]: True
```

Папа: **SETL**

Сын: **ABC**

Внук: **Python**

```
In [213]: [x ** 2 for x in range(10) if x % 2 == 1]
```

```
Out[213]: [1, 9, 25, 49, 81]
```

Компактная альтернатива комбинациям **map** и **filter**

```
In [214]: list(map(lambda x: x ** 2,
                  filter(lambda x: x % 2 == 1,
                         range(10))))
```

```
Out[214]: [1, 9, 25, 49, 81]
```

Могут быть вложенными

```
In [215]: [[j for j in range(5)] for i in range(5)]
```

```
Out[215]: [[0, 1, 2, 3, 4],  
           [0, 1, 2, 3, 4],  
           [0, 1, 2, 3, 4],  
           [0, 1, 2, 3, 4],  
           [0, 1, 2, 3, 4]]
```

```
In [216]: matrix = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]  
          [val for sublist in matrix for val in sublist]
```

```
Out[216]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Slicing

Слайсинг позволяет брать слайсы от последовательности через функцию `slice`. Может работать с любым объектом реализующим протокол последовательность (имплементированы методы `__getitem__()` and `__len__()`)

Синтаксис

```
In [217]: slice(stop)
           slice(start, stop, step)

-----
NameError                                 Traceback (most recent call last)
<ipython-input-217-95d7c145648e> in <module>
----> 1 slice(stop)
      2 slice(start, stop, step)

NameError: name 'stop' is not defined
```

```
In [218]: employee = ['Vasya', 'Pupkin', 'senior', '300k/ns']
NAME = slice(2)
POSITION = slice(2, 3)
SALARY = slice(3, None)
print(employee[NAME])
print(employee[SALARY])
print(SALARY)
```

```
['Vasya', 'Pupkin']
['300k/ns']
slice(3, None, None)
```

Теперь вы знаете:

- Как создать список любой вложенности
- Методы работы со списком
- Некоторые build-in функции
- Slicing

ВОПРОСЫ



~~Кортеж~~ Tuple

Литералы кортежа - простые скобки. Их можно опускать.

```
In [219]: point = 1, 2  
date = 'may', 22
```

Одноэлементный кортеж - всегда в скобках. В противном случае провоцирует трудноотлавливаемый баг.

Плохо:

```
In [220]: t = 'some',
x, = t
x
```

```
Out[220]: 'some'
```

Нормально:

```
In [221]: t = ('some', )
[x] = t
x
```

```
Out[221]: 'some'
```

По сути тоже самое, что и list, но неизменяемый.

Зачем?

- Обезопасить данные от изменения.
- В среднем работает быстрее списка
- Занимает меньше места.
- Могут быть ключем в словаре. Почему ?

```
In [222]: lst = [10, 20, 30]
tpl = (10, 20, 30)
print(lst.__sizeof__())
print(tpl.__sizeof__())
```

```
64
48
```

Сравнение кортежей происходит в лексикографическом порядке, длина учитывается только если одна последовательность является префиксом другой

```
In [223]: (1,2,3) < (1,2,4)
```

```
Out[223]: True
```

```
In [224]: (1, 2, 3, 4) < (1, 2, 4)
```

```
Out[224]: True
```

Конкатенировать можно через +, результат всегда новый объект.

Перевернуть tuple

```
In [225]: tuple(reversed((1, 2, 3)))
```

```
Out[225]: (3, 2, 1)
```

```
In [226]: (1, 2, 3)[::-1]
```

```
Out[226]: (3, 2, 1)
```

Зачем нужна "лишняя" функция reversed ?

- слайс всегда возвращает копию при работе сстроенными коллекциями
- reversed специализирован для разных типов, способ итерироваться с конца и не платить за это памятью. $O(1)$

enumerate

```
In [227]: my_list = ['apple', 'banana', 'grapes', 'pear']
for c, value in enumerate(my_list, 1):
    print(c, value)
```

```
1 apple
2 banana
3 grapes
4 pear
```

Предпочитай итерацию по объекту циклам со счётчиком. Ошибка на 1 в индексе --- это классика. Если же индекс требуется, помни про enumerate

```
In [ ]: # Плохо
for i in range(len(xs)) :
    x = xs[i]

# Лучше
for x in xs:
    ...

# Или
for i, x in enumerate(xs):
    ...
```

Рекомендую ознакомиться со всем списком встроенных функций (<https://docs.python.org/3/library/functions.html>)

tuple vs list

list

- Mutable
- Медленнее
- Семантически гомогенная последовательность
- Unhashable

tuple

- Immutable
- Быстрее
- Семантически гетерогенная структура данных
- Hashable

hashable

Объект будет `hashable` если у него есть некоторое хеш-значение, которое не меняется на протяжении жизни этого объекта (получают через метод `__hash__`), и его можно сравнить с другими объектами (реализован метод `__eq__` или `__cmp__`). Хешируемые объекты, которые считаются одинаковыми должны иметь одинаковое хэш значение. Пользовательские объекты по умолчанию реализуют метод `__hash__`.

```
In [229]: a = (1, 2, 3)
c = (1, 2, 3)
a == c
```

```
Out[229]: True
```

```
In [230]: a.__hash__() == c.__hash__()
```

```
Out[230]: True
```

```
In [231]: id(a) != id(c)
```

```
Out[231]: True
```

```
In[2]: class a:  
...:     pass  
...:  
In[3]: A = a()  
In[4]: A.a = 3  
In[5]: hash(a)  
Out[5]: -9223363245350239489  
In[6]: {A:'a'}  
Out[6]: {<__main__.a at 0x7feef567ea90>: 'a'}  
In[7]: A.a = 666  
In[8]: hash(a)  
Out[8]: -9223363245350239489
```

In [232]: `hash(-1) == hash(-2)`

Out[232]: True

Вопрос: какие ключи будут в словаре?

```
In [273]: d = {  
    False: 'False',  
    0: '0',  
}  
d
```

```
Out[273]: {False: '0'}
```

Namedtuple

Именованный кортеж - тип кортежа, специализированный на фиксированное множество полей. Каждый сохраненный в них объект может быть доступен через уникальный, удобный для чтения человеком идентификатор.

```
In [234]: from collections import namedtuple  
Car = namedtuple('Car' , 'color mileage')  
my_car = Car('red', 3812.4)  
my_car
```

```
Out[234]: Car(color='red', mileage=3812.4)
```

```
In [235]: my_car.color
```

```
Out[235]: 'red'
```

Индексы все еще доступны:

```
In [236]: my_car[0]
```

```
Out[236]: 'red'
```

Есть несколько удобных встроенных методов

```
In [237]: my_car._asdict()
```

```
Out[237]: OrderedDict([('color', 'red'), ('mileage', 3812.4)])
```

Метод `_replace()`. Создаёт поверхностную (shallow) копию кортежа и позволяет выборочно заменять некоторые поля:

```
In [238]: my_car._replace(color='blue')
```

```
Out[238]: Car(color='blue', mileage=3812.4)
```

Метод класса `_make()` может быть использован, чтобы создать новый экземпляр именованного кортежа из последовательности:

```
In [239]: Car._make(['red', 999])
```

```
Out[239]: Car(color='red', mileage=999)
```

Множественное присваивание и распаковка

```
In [240]: x, y = 10, 20
```

То же самое, но по-другому:

```
In [241]: x, y = 10, 20
x, y = (10, 20)
(x, y) = 10, 20
(x, y) = (10, 20)
```

Можно использовать на любом итерируемом объекте

```
In [242]: x, y = [10, 20]
x
```

```
Out[242]: 10
```

```
In [243]: x, y = 'hi'
x
```

```
Out[243]: 'h'
```

Работает с любым количеством объектов и даже с переменными:

```
In [244]: point = 10, 20, 30
x, y, z = point
print(x, y, z)
(x, y, z) = (z, y, x)
print(x, y, z)
```

```
10 20 30
30 20 10
```

Можно использовать asterisk:

```
In [245]: first, *_, last = range(4)
first, last
```

```
Out[245]: (0, 3)
```

```
In [246]: print(*[1], *[2], 3)
{*range(4), 4}
```

```
1 2 3
```

```
Out[246]: {0, 1, 2, 3, 4}
```

```
In [247]: first, *middle, last = range(4)
middle, type(middle)
```

```
Out[247]: ([1, 2], list)
```

Распаковка в звездочку - всегда список
[\(https://www.python.org/dev/peps/pep-3132/#acceptance\)](https://www.python.org/dev/peps/pep-3132/#acceptance)

Аналогично работает распаковка двумя звездочками **

```
In [248]: dict(**{'x': 1}, y=2, **{'z': 3})
```

```
Out[248]: {'x': 1, 'y': 2, 'z': 3}
```

В словаре свежие ключи будут перетирать уже существующие в словаре

```
In [249]: {'x': 1, **{'x': 2}}
```

```
Out[249]: {'x': 2}
```

```
In [250]: {**{'x': 2}, 'x': 1}
```

```
Out[250]: {'x': 1}
```



ВОПРОСЫ

Сет (множество)

неупорядоченная коллекция уникальных хешируемых объектов.

Где используют?

1. Удаление дубликатов из последовательности
2. Membership testing
3. Подсчет пересечений, объединений, разности и симметрической разности

NB!

- Сеты не поддерживают индексирование и слай싱
- Упорядоченной вставки у них нет

Как создать сет?

```
In [251]: container = set()
```

А еще сет задается фигурными скобками...

```
In [252]: lol_net = {}
type(lol_net)
```

```
Out[252]: dict
```

Но обязательно с элементами внутри! Множество имеет тот же литерал, что и словарь, но пустое множество с помощью литерала создать нельзя.

```
In [253]: lol = {"youth", "of", "the", "nation"}
print(lol, id(lol))
lol.add("me")
print(lol, id(lol))
```

```
{'of', 'the', 'youth', 'nation'} 139803673300104
{'the', 'youth', 'nation', 'me', 'of'} 139803673300104
```

А что будет, если подать не набор объектов, а одну-единственную строку?

```
In [254]: s = set("privet")
s
```

```
Out[254]: {'e', 'i', 'p', 'r', 't', 'v'}
```

Методы множеств

Возвращают True или False:

`set.isdisjoint(another_set)` - True, если `set` и `another_set` не имеют общих элементов.

`set.issubset(another_set)` - True, все элементы `set` принадлежат `another_set`.

`set.issuperset(another_set)` - True, все элементы `another_set` принадлежат `set`.

Вспоминая теорию множеств:

`set.union(set1, ...)` - объединение нескольких множеств.

`set.intersection(set1, ...)` - пересечение множеств.

`set.difference(other, ...)` - множество всех элементов `set`, не принадлежащих ни одному из `other`.

`set.symmetric_difference(other)` - все элементы исходных множеств, не принадлежащие одновременно обоим исходным множествам.

И еще один mutable - меняем жизнь множества!



`set.update(other, ...)` - объединение одного множества другим.

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

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

`set.symmetric_difference_update(other)` - множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.

`set.add(elem)` - добавляет элемент в множество.

`set.remove(elem)` - удаляет элемент из множества. `KeyError`, если такого элемента не существует.

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

`set.pop()` - удаляет первый элемент из множества. Так как множества не упорядочены, нельзя точно сказать, какой элемент будет первым.

`set.clear()` - очистка множества.

frozenset - неизменяемый тип множества.



```
In [255]: frozen = frozenset("diplome")
frozen.add("water")
```

```
-----  
AttributeError                                     Traceback (most recent call last)
<ipython-input-255-b00608177d90> in <module>
      1 frozen = frozenset("diplome")
----> 2 frozen.add("water")  
  
AttributeError: 'frozenset' object has no attribute 'add'
```


Итого

- Дали определение сету и его отмороженному родственнику
- Потрогали методы сетов и пустоту

Словарь - это неупорядоченная коллекция объектов с доступом по ключу.

Как инициализировать словарь?

1. С помощью литерала

```
In [256]: dictionary = {}  
# or  
dictionary = {'West World': 8.4, 'True Detective': 7.6}  
dictionary
```

```
Out[256]: {'West World': 8.4, 'True Detective': 7.6}
```

2. Функция dict()

```
In [257]: dictionary = dict()  
# or  
dictionary = dict([(1, 100), (2, 400)])  
dictionary
```

```
Out[257]: {1: 100, 2: 400}
```

3. Метод .fromkeys()

```
In [258]: dictionary = dict.fromkeys(["Leonardo", "Donatello",  
                                 "Raphael", "Michelangelo"], "ninja")  
dictionary
```

```
Out[258]: {'Leonardo': 'ninja',  
           'Donatello': 'ninja',  
           'Raphael': 'ninja',  
           'Michelangelo': 'ninja'}
```

4. Генератор словарей

```
In [259]: dictionary = {number: number**2 for number in range(0, 7, 2)}  
dictionary  
  
Out[259]: {0: 0, 2: 4, 4: 16, 6: 36}
```

Титул Ключа может быть присвоен

- единственному в своем роде
- хешируемому типу



Методы словарей

```
In [260]: [type(i) for i in dictionary.items()]
```

```
Out[260]: [tuple, tuple, tuple, tuple]
```

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

dict.keys() - возвращает ключи в словаре.

dict.values() - возвращает значения в словаре.

dict.pop(key[, default]) - удаляет ключ и возвращает значение. Если ключа нет, возвращает default (по умолчанию бросает исключение).

dict.popitem() - удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение KeyError. Помните, что словари неупорядочены.

dict.update([other]) - обновляет словарь, добавляя пары (ключ, значение) из other. Существующие ключи перезаписываются. Возвращает None (не новый словарь!).

Обсудили словари

- 4 способа создать словарь бесплатно
- Кому может быть присвоен титул Ключа
- Методы словарей
 - берем всё - .items()
 - ключи - .keys()
 - значения - .values()
 - какой вопрос, такой ответ - .get() / .setdefault()
 - попаем удаляем всех

Time complexity

List

```
In[2]: class a:  
...:     pass  
...:  
In[3]: A = a()  
In[4]: A.a = 3  
In[5]: hash(a)  
Out[5]: -9223363245350239489  
In[6]: {A:'a'}  
Out[6]: {<__main__.a at 0x7feef567ea90>: 'a'}  
In[7]: A.a = 666  
In[8]: hash(a)  
Out[8]: -9223363245350239489
```

Dict

```
In[2]: class a:  
...:     pass  
...:  
In[3]: A = a()  
In[4]: A.a = 3  
In[5]: hash(a)  
Out[5]: -9223363245350239489  
In[6]: {A:'a'}  
Out[6]: {<__main__.a at 0x7feef567ea90>: 'a'}  
In[7]: A.a = 666  
In[8]: hash(a)  
Out[8]: -9223363245350239489
```

Set - имплементация внутри Сpython намеренно очень похожа на dict

```
In[2]: class a:  
...:     pass  
...:  
In[3]: A = a()  
In[4]: A.a = 3  
In[5]: hash(a)  
Out[5]: -9223363245350239489  
In[6]: {A:'a'}  
Out[6]: {<__main__.a at 0x7feef567ea90>: 'a'}  
In[7]: A.a = 666  
In[8]: hash(a)  
Out[8]: -9223363245350239489
```

Файлы и работа с ними

Функция open

Текстовые и бинарные файлы в Python сильно отличаются.

Создать объект типа файл можно с помощью функции open, которая принимает один обязательный аргумент – путь к файлу:

```
In [263]: open('./data/test.txt')
```

```
Out[263]: <_io.TextIOWrapper name='./data/test.txt' mode='r' encoding='UTF-8'>
```

Аргументов у функции open - много, обратить внимание нужно на:

- **mode** - определяет в каком режиме будет открыт файл. Возможные значения:
 - "r", "w", "x", "a", "+"
 - "b", "t".
- для текстовых файлов
 - **encoding**
 - **errors**

In [264]:

	r	r+	w	w+	a	a+
read	+	+		+		+
write		+	+	+	+	+
write after seek		+	+	+		
create			+	+	+	+
truncate				+	+	
position at start	+	+	+	+		
position at end					+	+

```
File "<ipython-input-264-c0fa45e7d330>", line 1
| r  r+  w  w+  a  a+
^
```

SyntaxError: invalid syntax

Создать новый текстовый файл в системной кодировке и открыть его для записи:

In []: `open("./data/test.txt", "x") # failing if the file already exists`

Методы работы с файлами

Чтение

Метод `read` читает не более `n` символов из файла

```
In [142]: file_handle = open('./data/test.txt')
file_handle.read(7)
```

```
Out[142]: 'line1\n'
```

Методы `readline` и `readlines` читают одну или все строчки соответственно. Можно указать максимальное количество символов, которые надо прочитать:

```
In [143]: file_handle = open('./data/test.txt')
print(len(file_handle.readline()))

file_handle.readlines()
```

```
6
```

```
Out[143]: ['line2\n', 'line3\n']
```

Запись

```
In [151]: file_handle = open("./data/example.txt", "w")
file_handle.write('someinformation')
```

```
Out[151]: 15
```

Неявного добавления символа переноса строки, как в `print` - нет!

Записать последовательность строк можно с помощью метода `writelines`:

```
In [152]: file_handle.writelines(['spam', 'egg'])
file_handle.close()
open("./data/example.txt", "r").readlines()
```

```
Out[152]: ['someinformationspamegg']
```

И еще немного методов работы с файлом

```
In [153]: file_handle = open("./data/example.txt", "r+")
file_handle.fileno() # file descriptor
```

```
Out[153]: 46
```

```
In [154]: file_handle.tell() # file object's current position in bytes
```

```
Out[154]: 0
```

```
In [155]: file_handle.seek(8)
file_handle.tell()
```

```
Out[155]: 8
```

```
In [156]: file_handle.write("something unimportant")
file_handle.flush() # Flush the write buffers of the stream
file_handle.close()
```

Файл всегда нужно закрывать

А сделать это удобно можно с помощью контекстного менеджера:

```
In [157]: with open("./data/example.txt", "r+") as ouf:  
    ...  
    # do your stuff here and dont worry about file closing
```

Вся правда о файлах

В UNIX - все файл: директории, жесткий диск, сетевые устройства, пайплайны, stdin/stdout и тд.

И работая с файлами в shell мы как правило работаем с **inode**

- Inode - структура данных, которая является репрезентацией файла (хранит метаинформацию)
- Index node (maybe?)
- Хранится на диске, указывает на фактическое местоположение файла

Хранит:

- группу и имя владельца
- тип: обычный, директория, символьное или блочное устройство, FIFO pipe
- Права доступа
- время доступа: к файлу, файл изменен, inode'a изменена.
- Количество жестких ссылок к файлу
- Адреса блоков диска, хранящих информацию
- Размер файла

Не хранит путь до файла!

Файловый дескриптор – это неотрицательное целое число. Когда мы открываем существующий файл и создаем новый файл, ядро возвращает процессу файловый дескриптор.

При запуске программы в оболочке открывается три дескриптора 0, 1 и 2. По умолчанию с ними связаны следующие файлы:

По умолчанию UNIX-шеллы связывают файловый дескриптор 0 со стандартным вводом процесса (терминал), файловый дескриптор 1 – со стандартным выводом (терминал), и файловый дескриптор 2 – со стандартной ошибкой (то есть то куда выводятся сообщения об ошибках). Это соглашение соблюдается многими UNIX-шеллами и многими приложениями – и в ни коем случае не является составной частью ядра.

Почему важно закрывать файловый дискриптор ?

- Отправляет программу в руки сборщика мусора
- Может замедлить программу. Слишком много открытых вещей - большой расход RAM
- Большинство изменений в файле применяется только после закрытия файла.
- Теоретически вы можете упереться в лимит ОС на открытые файлы
- Windows считает открытые файлы заблокированными, так что, например, другой скрипт на питоне не сможет их прочитать.
- Дурной тон

**Все это, конечно, прекрасно и
обязательно нужно запомнить...**

```
In [274]: help(str)
# https://docs.python.org/3/
```

Help on class str in module builtins:

```
class str(object)
| str(object='') -> str
| str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or
errors is specified, then the object must expose a data buffer
that will be decoded using the given encoding and error handler.
Otherwise, returns the result of object.__str__() (if defined)
or repr(object).
encoding defaults to sys.getdefaultencoding().
errors defaults to 'strict'.
```

Methods defined here:

```
__add__(self, value, /)
    Return self+value.

__contains__(self, key, /)
    Return key in self.

__eq__(self, value, /)
    Return self==value.

__format__(self, format_spec, /)
    Return a formatted version of the string as described by format_spec.

__ge__(self, value, /)
    Return self>=value.

__getattribute__(self, name, /)
    Return getattr(self, name).

__getitem__(self, key, /)
    Return self[key].

__getnewargs__(...)

__gt__(self, value, /)
    Return self>value.

__hash__(self, /)
    Return hash(self).

__iter__(self, /)
    Implement iter(self).

__le__(self, value, /)
    Return self<=value.

__len__(self, /)
    Return len(self).

__lt__(self, value, /)
    Return self<value.

__mod__(self, value, /)
    Return self%value.

__mul__(self, value, /)
    Return self*value.

__ne__(self, value, /)
    Return self!=value.

__repr__(self, /)
    Return str(self).  
In [275]:
```




In []: