

Лекция 3

Язык программирования Python.

План занятия

- Коллекция
- Индексация
- tuple
- list
 - Срезы
 - Списковые включения
- Операции над списками

--

- Hash function
- Словарь - Dict
 - definition
- Множество - Set
- Генераторы

План занятия

- Общее представление

Инициализация аргументов функции

В случае инициализации аргументов функции мутабельными объектами, важно учитывать что. Инициализация будет произведена один раз а не каждый раз при вызове функции

Инициализация аргументов функции

```
In [1]: def foo(l=[]):  
...:     l.append(1)  
...:
```

```
In [2]: def foo(l=[]):  
...:     l.append(1)  
...:     print(l)  
...:
```

```
In [3]: foo()  
[1]
```

```
In [4]: foo()  
[1, 1]
```

```
In [5]: foo()  
[1, 1, 1]
```

```
In [6]: foo([1,2,3])  
[1, 2, 3, 1]
```

Массив

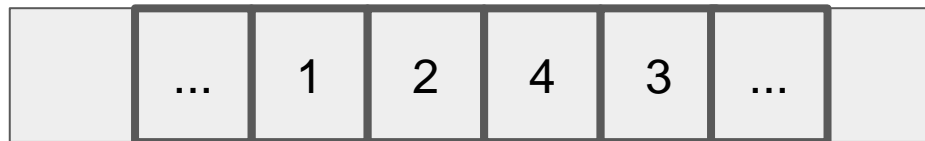
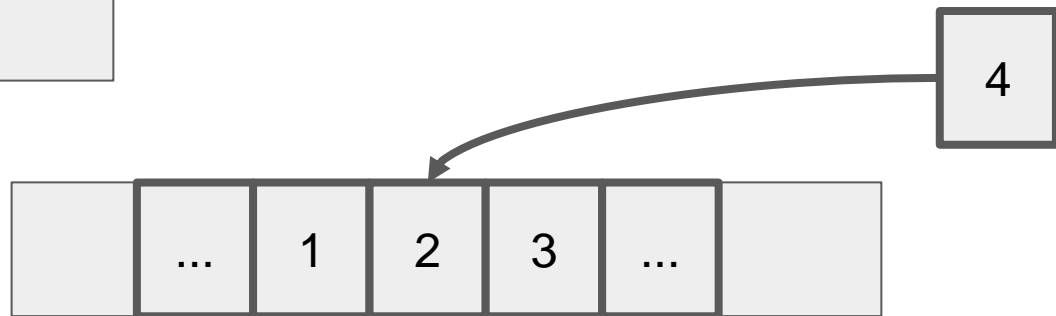
Фундаментальная структура данных, позволяющая хранить некоторый объем непересекающихся данных в непрерывном куске оперативной памяти. Обычно массивы представляют к каждому отдельной сущности массива через порядковый индекс - это операция называется **индексация**. Поддерживается большинством языков программирования. Python не исключение.

Динамический массив

Обычно при создании массива предполагается, что массив будет иметь фиксированную длину, которая не меняется. Соответственно стандартный массив не поддерживает операции вставки и удаления элемента. Поэтому были введены специализированный тип данных как **динамический массив**.

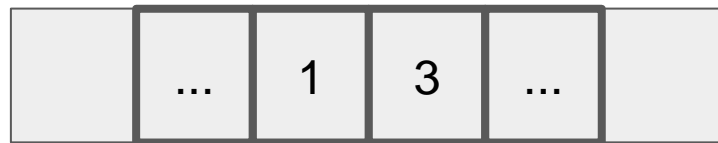
Массив

RAM



Массив

RAM



Коллекции

- **Коллекция** — программный объект, содержащий в себе, тем или иным образом, набор значений одного или различных типов, и позволяющий обращаться к этим значениям.
- **Коллекция** позволяет записывать в себя значения и извлекать их.
- **Назначение коллекции** — служить хранилищем объектов и обеспечивать доступ к ним.

Индексация

Некоторые коллекции поддерживают операцию индексации

В python операция обозначается [...] - аргументом операции является индекс, который указывает какой элемент коллекции нужно взять

Синтаксис:

```
collection_name[index]
```

где collection_name - переменная содержащая ссылку на коллекцию

index - указатель на элемент коллекции

Индексация

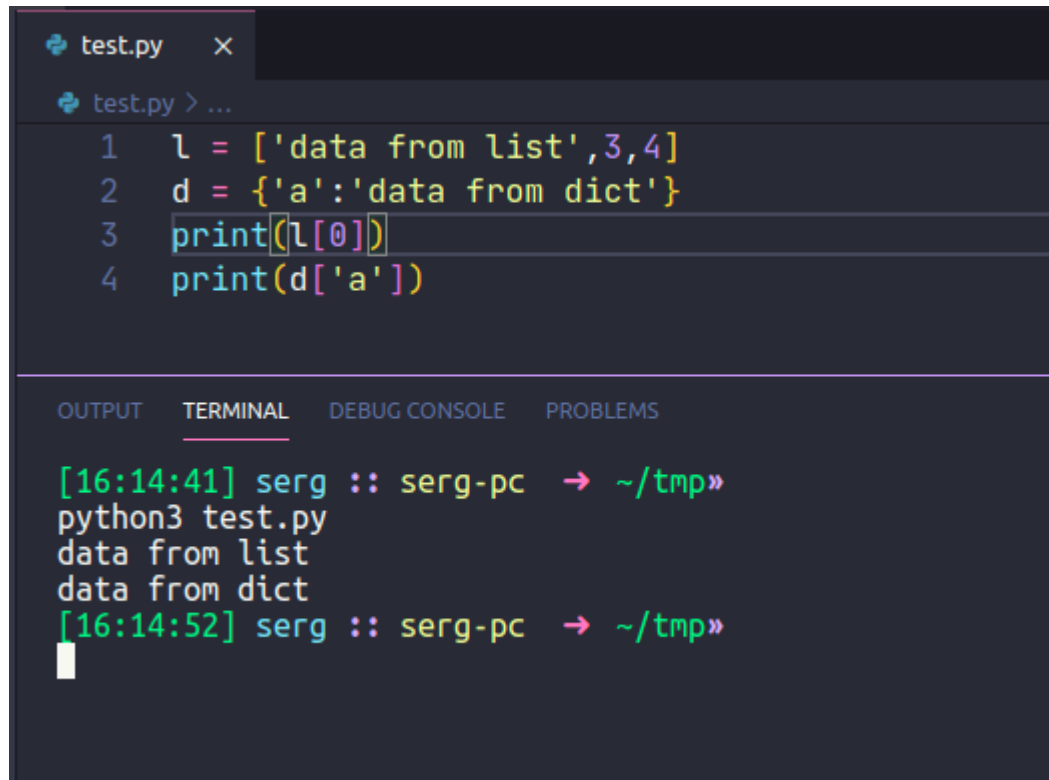
Для коллекций, сохраняющих порядок элементов, таких как **list**, **tuple**, **range** - индексом является порядковый номер элемента в коллекции.

ИНДЕКСАЦИЯ В PYTHON ДЛЯ ТАКИХ КОЛЛЕКЦИЙ НАЧИНАЕТСЯ С 0!

Также в python поддерживаются отрицательные индексы (см. примеры).

Для **dict**, **set**, ... - индексом является значение ключа элемента (об этом на др лекции).

Индексация



The image shows a code editor window with a file named `test.py`. The code defines a list `l` and a dictionary `d`, then prints the first element of the list and the value for the key 'a' in the dictionary. Below the code, the terminal output shows the script being executed successfully, printing the expected results.

```
test.py  X
test.py > ...
1  l = ['data from list',3,4]
2  d = {'a':'data from dict'}
3  print(l[0])
4  print(d['a'])
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
[16:14:41] serg :: serg-pc  → ~/tmp»
python3 test.py
data from list
data from dict
[16:14:52] serg :: serg-pc  → ~/tmp»
```

Python - collection package

`collections` — Container datatypes

Source code: [Lib/collections/__init__.py](#)

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, `dict`, `list`, `set`, and `tuple`.

<code>namedtuple()</code>	factory function for creating tuple subclasses with named fields
<code>deque</code>	list-like container with fast appends and pops on either end
<code>ChainMap</code>	dict-like class for creating a single view of multiple mappings
<code>Counter</code>	dict subclass for counting hashable objects
<code>OrderedDict</code>	dict subclass that remembers the order entries were added
<code>defaultdict</code>	dict subclass that calls a factory function to supply missing values
<code>UserDict</code>	wrapper around dictionary objects for easier dict subclassing
<code>UserList</code>	wrapper around list objects for easier list subclassing
<code>UserString</code>	wrapper around string objects for easier string subclassing

Deprecated since version 3.3, will be removed in version 3.10: Moved [Collections Abstract Base Classes](#) to the `collections.abc` module. For backwards compatibility, they continue to be visible in this module through Python 3.9.

<https://docs.python.org/3.8/library/collections.html>

Python built-in collection

Sequence Types — list, tuple, range

There are three basic sequence types: lists, tuples, and range objects. Additional sequence types tailored for processing of binary data and text strings are described in dedicated sections.

<https://docs.python.org/3.8/library/stdtypes.html#sequence-types-list-tuple-range>

tuple

```
test.py x
test.py > ...
1 l1 = 1,2,3
2 l2 = (1,2,3)
3 print(l1)
4 print(l2)
5
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
[15:10:10] serg :: serg-pc → ~/tmp»
cd /home/serg/tmp ; /usr/bin/env /usr/bin/python3
g/tmp/test.py
(1, 2, 3)
(1, 2, 3)
[15:10:14] serg :: serg-pc → ~/tmp»
```

```
test.py x
test.py > ...
1 l1 = 1,2,3
2 l1[0] = 123
```

Exception has occurred: TypeError
'tuple' object does not support item assignment
File "/home/serg/tmp/test.py", line 2, in <module>
l1[0] = 123

<https://docs.python.org/3.8/library/stdtypes.html#tuple>

list

```
test.py x
test.py > ...
1 l = [1,2,3,4]
2 print(l)

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

[15:12:37] serg :: serg-pc → ~/tmp»
cd /home/serg/tmp ; /usr/bin/env /usr/bin/python3
g/tmp/test.py
[1, 2, 3, 4]
[15:12:43] serg :: serg-pc → ~/tmp»
```

```
test.py x
test.py > ...
1 l = [1,2,3,4]
2 print(l)
3 l[2] = 3.14
4 print(l)

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

[15:15:11] serg :: serg-pc → ~/tmp»
cd /home/serg/tmp ; /usr/bin/env /usr/bin/python3
[1, 2, 3, 4]
[1, 2, 3.14, 4]
[15:15:16] serg :: serg-pc → ~/tmp»
```

range - объект

Позволяет создавать генератор последовательности с заданными стартовым конечным (не включается) и шагом значениями. При этом последовательность не генерируется моментально.

range(begin, end, step)

```
In [16]: [1,2,3,4]
Out[16]: [1, 2, 3, 4]
```

```
In [17]: range(10)
Out[17]: range(0, 10)
```

```
In [18]: range(1, 10)
Out[18]: range(1, 10)
```

```
In [19]: range(1, 10, 2)
Out[19]: range(1, 10, 2)
```

```
In [20]: list(range(10))
Out[20]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [21]: list(range(1, 10))
Out[21]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [22]: list(range(1, 10, 2))
Out[22]: [1, 3, 5, 7, 9]
```

range - объект

```
In [23]: range(-100)
Out[23]: range(0, -100)
```

```
In [27]: list(range(10, 1, -1))
Out[27]: [10, 9, 8, 7, 6, 5, 4, 3, 2]
```

list

Operation	Result
<code>s[i] = x</code>	item <code>i</code> of <code>s</code> is replaced by <code>x</code>
<code>s[i:j] = t</code>	slice of <code>s</code> from <code>i</code> to <code>j</code> is replaced by the contents of the iterable <code>t</code>
<code>del s[i:j]</code>	same as <code>s[i:j] = []</code>
<code>s[i:j:k] = t</code>	the elements of <code>s[i:j:k]</code> are replaced by those of <code>t</code> (1)
<code>del s[i:j:k]</code>	removes the elements of <code>s[i:j:k]</code> from the list
<code>s.append(x)</code>	appends <code>x</code> to the end of the sequence (same as <code>s[len(s):len(s)] = [x]</code>)
<code>s.clear()</code>	removes all items from <code>s</code> (same as <code>del s[:]</code>) (5)
<code>s.copy()</code>	creates a shallow copy of <code>s</code> (same as <code>s[:]</code>) (5)
<code>s.extend(t)</code> or <code>s += t</code>	extends <code>s</code> with the contents of <code>t</code> (for the most part the same as <code>s[len(s):len(s)] = t</code>)
<code>s *= n</code>	updates <code>s</code> with its contents repeated <code>n</code> times (6)
<code>s.insert(i, x)</code>	inserts <code>x</code> into <code>s</code> at the index given by <code>i</code> (same as <code>s[i:i] = [x]</code>)
<code>s.pop([i])</code>	retrieves the item at <code>i</code> and also removes it from <code>s</code> (2)
<code>s.remove(x)</code>	remove the first item from <code>s</code> where <code>s[i] == x</code> (3)
<code>s.reverse()</code>	reverses the items of <code>s</code> in place (4)

len

Встроенная функция [len](#) позволяет выяснить текущую длину коллекции тип коллекции не имеет значения.

```
In [12]: l = [1,2,3]
```

```
In [13]: m = {'key1':'value1', 'key2':'value2'}
```

```
In [14]: len(l)
```

```
Out[14]: 3
```

```
In [15]: len(m)
```

```
Out[15]: 2
```

append

Добавляет новый элемент в конец списка. Довольно эффективная процедура, так как при этом список **почти всегда** не реаллоцирует память.

insert

В случае если необходимо вставить элемент в середину или в начало списка, можно воспользоваться этой процедурой, **но** в этом случае список память будет реалоцирована, что может привести к потере производительности. Реалокация - процесс перераспределения памяти.

`list.insert(i, x)`

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

index

Поиск элемента по значению в последовательности - вернет индекс элемента в списке. Если элемента нет то будет порождено исключение

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

Return zero-based index in the list of the first item whose value is equal to *x*. Raises a **ValueError** if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

remove

Удаление элемента из списка по значению. Также в случае отсутствия такого элемента в списке будет порождено исключение

```
list.remove(x)
```

Remove the first item from the list whose value is equal to `x`. It raises a `ValueError` if there is no such item.

pop

Извлечь элемент из списка и вернуть его. В случае если мы извлекаем не последний элемент процесс также приведет к релокации памяти.

`list.pop(i)`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

Списки арифметика (конкатенация)

Списки поддерживают операцию сложения

```
In [1]: l = [1,2,3,4]

In [2]: l1 = [5,6,7,8]

In [3]: l2 = l + l1

In [4]: l2
Out[4]: [1, 2, 3, 4, 5, 6, 7, 8]

In [5]: l += l1

In [6]: l
Out[6]: [1, 2, 3, 4, 5, 6, 7, 8]

In [7]:
```

Индексация списков

Для списков применима индексация с отрицательными значениями индекса при этом значению -1 будет соответствовать последний элемент списка, -2 предпоследний то есть

```
In [7]: l = [1,2,3]

In [8]: l[-1]
Out[8]: 3

In [9]: l[-1] == l[len(l) - 1]
Out[9]: True
```

Индексация списков

При обращении по индексу превосходящему длину списка будет сгенерировано исключение

```
In [10]: l[100]
```

```
-----  
IndexError                                Traceback (most recent call last)  
<ipython-input-10-e2a0c2623844> in <module>  
----> 1 l[100]
```

```
IndexError: list index out of range
```

Создание списков

Список можно создать несколькими способами

1. Явно
2. Списковые включения
3. Ключевое слово `list`

Явно

Через использование конструкции

[element1, element2, ..., elementN]

```
In [16]: [1,2,3,4]  
Out[16]: [1, 2, 3, 4]
```

Списковые включения

Более короткий синтаксис, который позволяет создать новый список на основе значений существующего списка.

```
[item for item in collection if condition ]
```

Условный оператор может отсутствовать.

Списковые включения

```
test.py x
test.py > ...

1  # How to create list
2  #1. loops
3  l = [] # empty list
4  for i in range(10):
5      l.append(i)
6
7  print(l)
8
9  # list comprehension
10 l = [i for i in range(10)]
11 print(l)
```

Списковые включения

```
test.py x
test.py > ...

1  # How to create list
2  #1. loops
3  l = [] # empty list
4  for i in range(10):
5      l.append(i)
6
7  print(l)
8
9  # list comprehension
10 l = [i for i in range(10)]
11 print(l)
```

Списковые включения

```
1 # inner list comprehension
2 l = [[j for j in range(i)] for i in range(10)]
3 for row in l:
4     print(row)
```

OUTPUT

TERMINAL

DEBUG CONSOLE

PROBLEMS

```
[17:07:56] serg :: serg-pc → ~/tmp»
python3 test.py
[]
[0]
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Списковые включения

```
test.py > ...  
1 # inner list comprehension  
2 l = [i for i in range(10) if i % 2 == 0]  
3 print(l)
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
[17:09:16] serg :: serg-pc → ~/tmp»  
python3 test.py  
[0, 2, 4, 6, 8]  
[17:09:18] serg :: serg-pc → ~/tmp»  
█
```

Ключевое слово list

Позволяет создавать списки в том числе десериализованные из другого типа последовательностей

```
In [34]: list('abcd')
Out[34]: ['a', 'b', 'c', 'd']

In [35]: list({'1':'2'})
Out[35]: ['1']

In [36]: list((1,2,3))
Out[36]: [1, 2, 3]
```

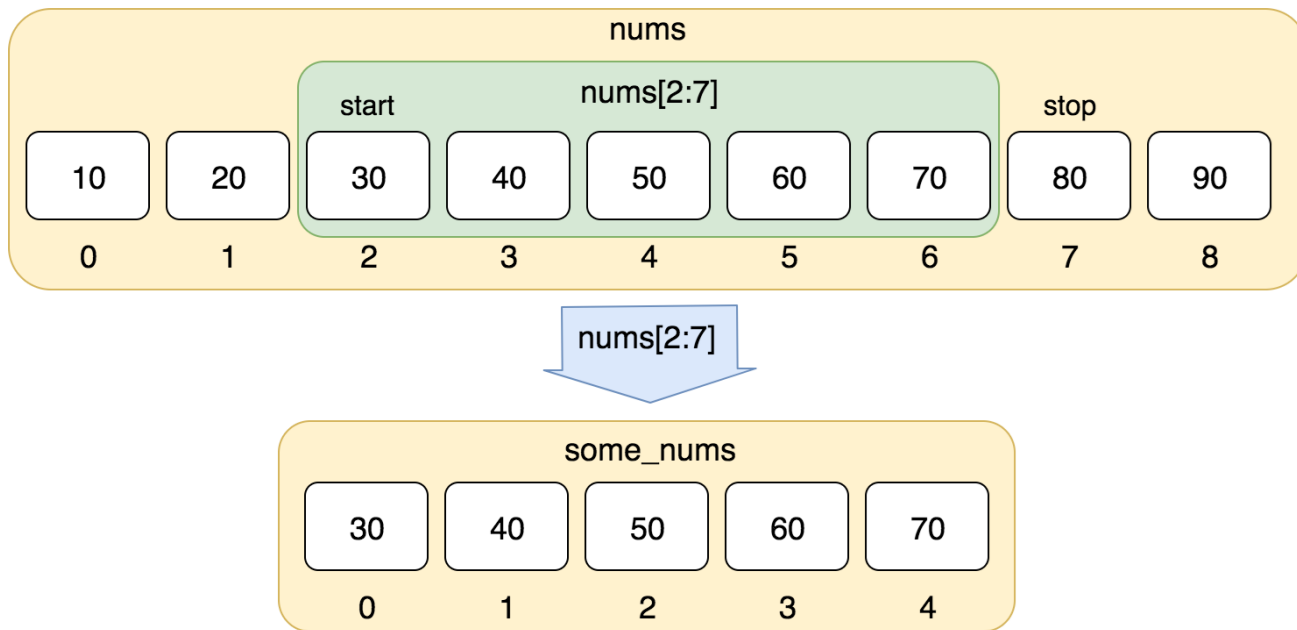
Срезы (slice)

Расширенная операция индексации, над сохраняющими порядок коллекциями, которая позволяет делать выборки из них. Параметры задаются значениями индексов правым (**который включается в выборку**) и левым (**не включается в выборку**). Синтаксис

`collection_name[start:stop:step]` -> вернет список в котором первым элементом будет элемент `collection_name[start]`

Значение индексов могут быть < 0 . Значение `step` - шаг может отсутствовать по умолчанию он равен 1.

Срезы (slice)



Срезы (slice)

```
1 l = [i for i in range(10)]
2
3 print(l)
4 print(l[1:7])
5 print(l[4:])
6 print(l[:5])
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
[16:43:49] serg :: serg-pc → ~/tmp»
python3 test.py
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6]
[4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4]
[16:43:50] serg :: serg-pc → ~/tmp»
```


Tricks

Самый простой способ перевернуть список

```
In [7]: l
Out[7]: [1, 2, 3, 4, 5, 6, 7, 8]

In [8]: l[::-1]
Out[8]: [8, 7, 6, 5, 4, 3, 2, 1]
```

Взять каждый второй элемент

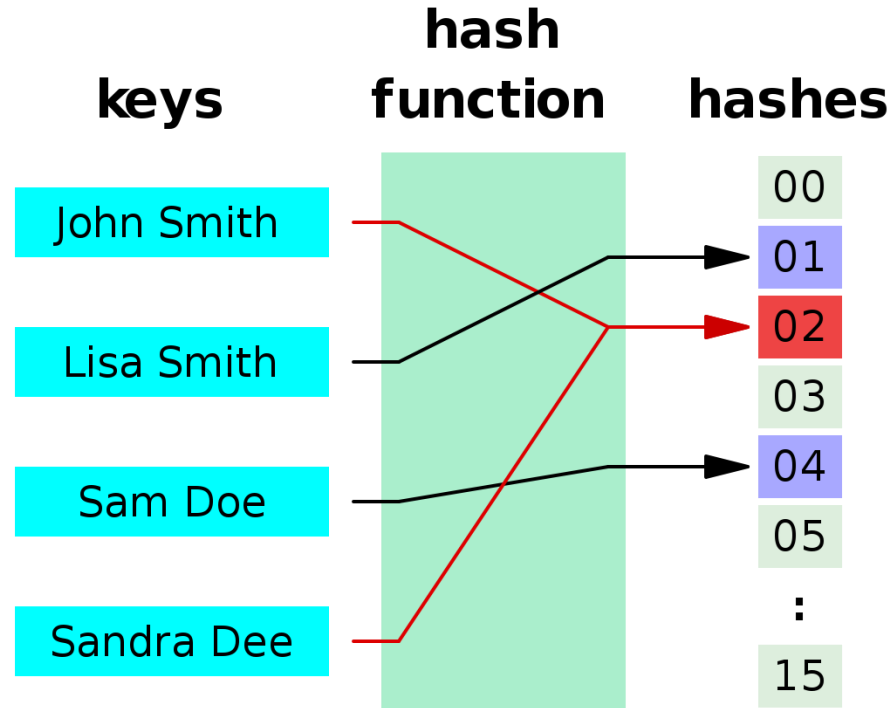
```
In [9]: l[::2]
Out[9]: [1, 3, 5, 7]
```

Hash function

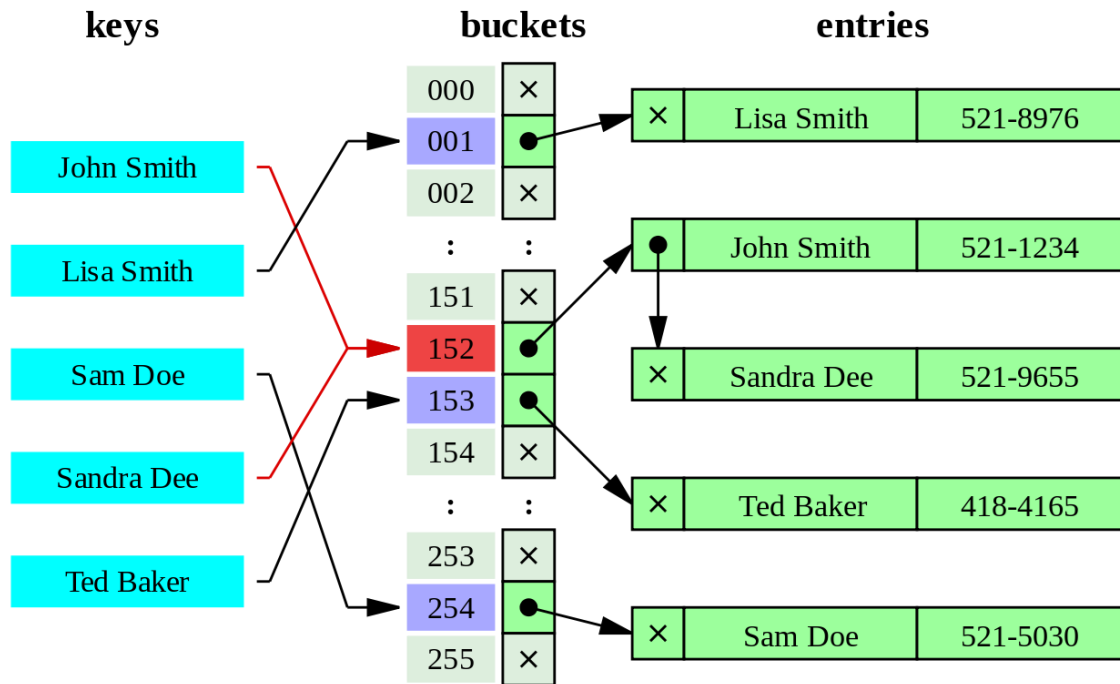
Неоднозначная функция принимающая на вход некоторые данные и возвращающая некоторое целое число $\in [0, \dots, N]$. Хорошая хеш-функция должна удовлетворять двум свойствам:

- быстрое вычисление
- минимально количество коллизий

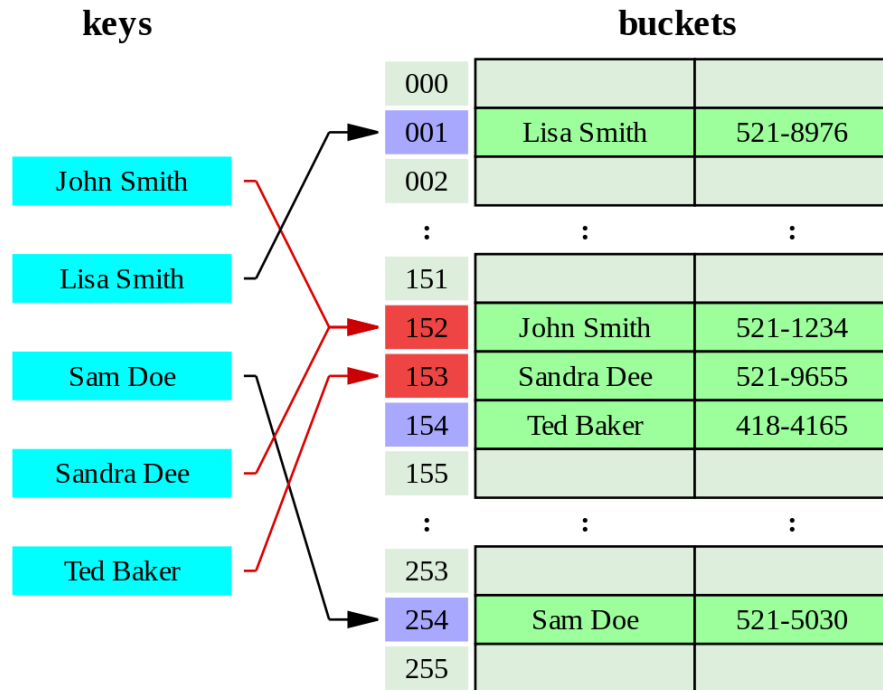
Hash function



Hash function (коллизия)



Hash function (коллизия)



Hash table

Еще называют хеш таблицей(дословный перевод), ассоциативный массив, словарь... Структура, которая не сохраняет порядок элементов при вставки, НО обеспечивает при этом быстрый доступ, быстрое удаление и вставку. В худшем и среднем случае работы этих функций производится за константное время в худшем за линейное.

Hash table

Python, как и почти все языки программирования дает возможность программисту создавать такие структуры. Ключами при этом могут быть только те типы данных, которые поддерживают операцию хеширования - значениями могут быть любые типы данных.

Hash table (python)

Словарь можно определить несколькими способами

- Явно
- Dict comprehension
- через цикл
- ключевое слово dict

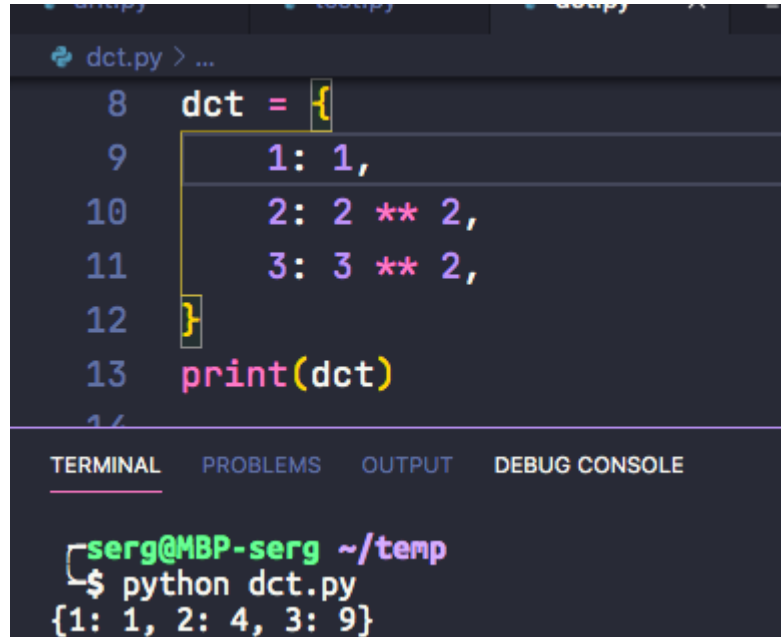
Hash table (python)

```
dct.py > ...
1  hash_table = {}
2  hash_table["key1"] = "value"
3  print(hash_table)
```

TERMINAL PROBLEMS 2 OUTPUT DEBUG CONSOLE

```
serg@MBP-serg ~/temp
$ python dct.py
{'key1': 'value'}
serg@MBP-serg ~/temp
$
```

Python **dict** obj definition 1



The image shows a code editor window with a file named `dct.py`. The code defines a dictionary `dct` with three key-value pairs: `1: 1`, `2: 2 ** 2`, and `3: 3 ** 2`. The dictionary is then printed. Below the code editor, the terminal output shows the command `python dct.py` being executed, resulting in the output `{1: 1, 2: 4, 3: 9}`.

```
dct.py > ...  
8  dct = {  
9      1: 1,  
10     2: 2 ** 2,  
11     3: 3 ** 2,  
12 }  
13 print(dct)  
14
```

serg@MBP-serg ~/temp
\$ python dct.py
{1: 1, 2: 4, 3: 9}

Python **dict** obj definition 2

```
dct.py > ...
14
15  dct = {i: i**2 for i in range(1,4)}
16  print(dct)
17
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
serg@MBP-serg ~/temp
$ python dct.py
{1: 1, 2: 4, 3: 9}
```

Python **dict** obj definition 3

```
21  dct = {}
22  for i in range(1, 4):
23      key = i
24      value = i ** 2
25      dct[key] = value
26  print(dct)
```

TERMINAL

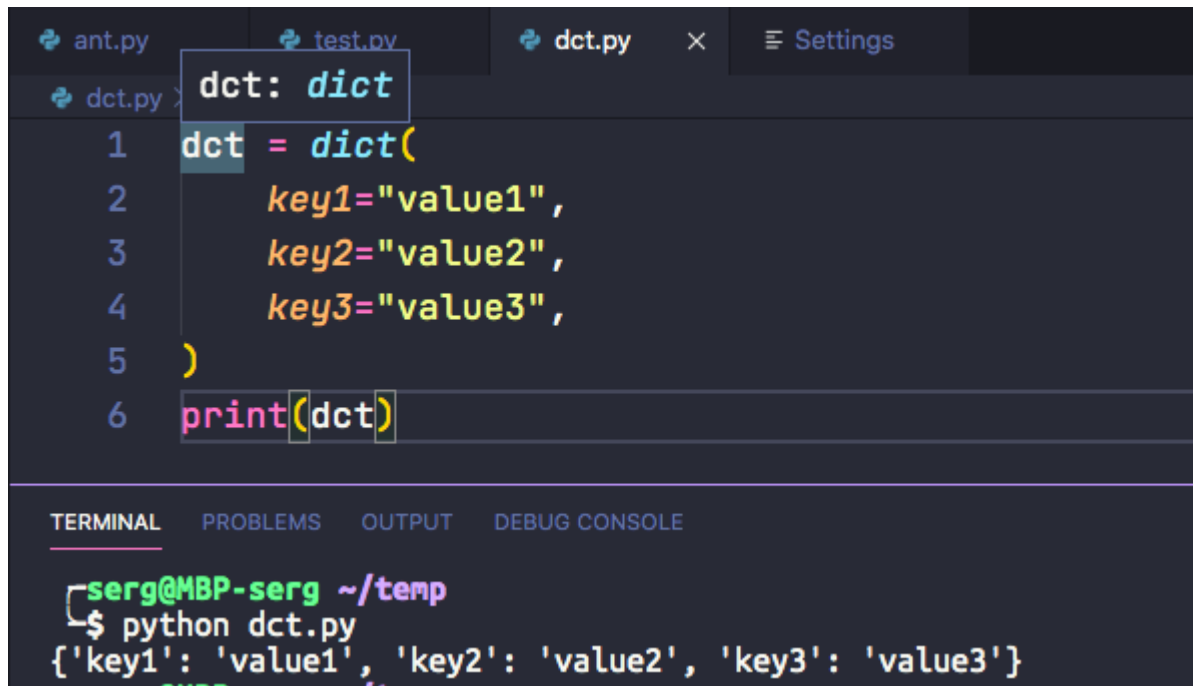
PROBLEMS

OUTPUT

DEBUG CONSOLE

```
serg@MBP-serg ~/temp
$ python dct.py
{1: 1, 2: 4, 3: 9}
```

Python **dict** obj definition 3



The image shows a code editor with three tabs: `ant.py`, `test.py`, and `dct.py`. The `dct.py` tab is active, displaying the following Python code:

```
1 dct = dict(  
2     key1="value1",  
3     key2="value2",  
4     key3="value3",  
5 )  
6 print(dct)
```

A tooltip is visible over the `dict` function call on line 1, showing `dct: dict`. Below the code editor is a terminal window with the following output:

```
serg@MBP-serg ~/temp  
$ python dct.py  
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
```

Python **dict** obj definition 4

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> f = dict({'one': 1, 'three': 3}, two=2)
>>> a == b == c == d == e == f
True
```

<https://docs.python.org/3/library/stdtypes.html#typesmapping>

Dict - операции

dict.clear() - очищает словарь.

dict.copy() - возвращает копию словаря.

dict.get(key[, default]) - возвращает значение ключа, но если его нет, не бросает исключение, а возвращает default (по умолчанию None).

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

dict.keys() - возвращает ключи 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 - операции

dict.setdefault(key[, default]) - возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ со значением default (по умолчанию None).

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

dict.values() - возвращает значения в словаре.

Dict (проход)

```
dct.py > ...
1  dct = {i: i**2 for i in range(1,4)}
2  print("-----")
3  for k in dct:
4      print(k, dct[k])
5  print("-----")
6  for k in dct.keys():
7      print(k, dct[k])
8  print("-----")
9  for v in dct.values():
10     print(v)
11  print("-----")
12  for k, v in dct.items():
13     print(k, v)
```

serg@MBP-serg ~/temp
\$ python dct.py

```
-----
1 1
2 4
3 9
-----
1 1
2 4
3 9
-----
1
4
9
-----
1 1
2 4
3 9
```

Dict - ключи

- immutable objects
- hashable
- comparable \Rightarrow obj1 == obj2

Dict - ключи

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Dict - ключи

```
In [2]: d = {[1,2,3]: 1}
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-2-7e7847a84a44> in <module>()  
----> 1 d = {[1,2,3]: 1}
```

```
TypeError: unhashable type: 'list'
```

```
In [3]: d = {(1,2,3): 1}
```

```
In [4]: d
```

```
Out[4]: {(1, 2, 3): 1}
```

Set - множество

Структура данных очень похожая на словарь, но в отличие от словаря хранит только ключи. Ключи уникальны. Сам тип данных реализует теоретико-множественные операции.

Множество (Set)

```
In [13]: s1 = set((1,2,3))
```

```
In [14]: s2 = {1,2,3}
```

```
In [15]: s1
```

```
Out[15]: {1, 2, 3}
```

```
In [16]: s2
```

```
Out[16]: {1, 2, 3}
```

```
In [17]: s1 == s2
```

```
Out[17]: True
```

set операции

len(s) - вернет длину множества

x in s - проверить наличие элемента в множестве

x not in s - проверить отсутствие элемента в множестве

isdisjoint(other) - Верните **True**, если множество не имеет общих элементов с другими. Множества не пересекаются тогда и только тогда, когда их пересечение - пустое множество.

issubset(other) - наоборот

set операции

set <= other - Является ли каждый ли элемент в наборе состоит из другого.

set < other - Тоже самое но строго

set >= other

set > other

union(*others) \Rightarrow **set | other | ...** - Объединение множеств в одно результат множество

intersection(*others) \Rightarrow **set & other & ...** Пересечение всех множеств, то есть результат это множество из элементов общих для всех

difference(*others) \Rightarrow **set - other - ...**

set ^ other - Вернет новое множество элементов, которые содержатся только в одном из множеств, но не в обоих

copy() Верните неглубокую копию множества.

Множество (Set) операции

```
In [31]: 1 in s1
```

```
Out[31]: True
```

```
In [32]: s2.add(4)
```

```
In [33]: s2
```

```
Out[33]: {1, 2, 3, 4}
```

```
In [34]: s1 ^ s2
```

```
Out[34]: {4}
```

```
In [35]: s1 | s2
```

```
Out[35]: {1, 2, 3, 4}
```

```
In [36]: s1 & s2
```

```
Out[36]: {1, 2, 3}
```

Определение

Generator functions allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop (<https://wiki.python.org/moin/Generators>).

При создании генератора нет необходимости выделять такое количество памяти, которое необходимо для хранения всего массива данных.

Создание

```
In [1]: gen = (i for i in range(10))
```

```
In [2]: type(gen)
```

```
Out[2]: generator
```

```
In [3]: for i in gen:  
...:     print(i)  
...:
```

0

1

2

3

4

5

6

7

8

9

Создание

```
class Arifmetic:
    def __init__(self, end, start=0, step=1):
        self.start = start
        self.n = end
        self.step = step
        self.cnt = 1

    def __next__(self):
        if self.cnt < self.n:
            self.cnt += 1
            self.start += self.step
            return self.start
        else:
            raise StopIteration()

    def __iter__(self):
        return self

a = Arifmetic(100, 2, 2)
print(sum(a))
```

Ключевое слово yield

```
1 def gen(start, end, step):
2     while start < end:
3         yield start
4         start += step
5
6
7 g = gen(1, 10, 1)
8 for i in g:
9     print(i)
10
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
[13:10:56] serg :: serg-pc → ~/tmp»
python test.py
```

```
1
2
3
4
5
6
7
8
9
```

```
[13:10:57] serg :: serg-pc → ~/tmp»
```

Домашнее задание

1. Отсортируйте список случайной длины в зависимости от мода по возрастанию и по убыванию
2. Найдите сумму всех чисел меньше 1000, кратных 3 или 5 с помощью функции генератора
3. Запишите в словарь по ключам от 1 до 10, список чисел (подается на входе), которые делятся на соответствующие ключи