KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 1 урок > Основы программирования Python. Типы и структуры данных

> Оглавление 1 урока

Скачать Ноутбук

Скачать в PDF

> Переменные

Переменная — это простейшая структура данных, в которую можно что-то передать.

Переменную создать просто. Необходимо присвоить некоторому идентификатору (названию) какое-то значение через знак `=`.

Хорошая практика в Python называть переменные snake_case методом, то есть использовать только буквы нижнего регистра, а слова разделять через нижние подчеркивания. Это помогает сделать код более читабельным. Другие практики написания переменных в Python нежелательны.

```
# так хорошо
snake_case_variable = 10000

# так лучше не надо
camelCase = -10
```

Подробнее про соглашения в написании переменных тут.

Комментирование

К коду можно добавлять пояснения в виде комментариев.

Чтобы создать комментарий, перед текстом нужно добавить символ `#`.

Ошибки при работе с переменными

- Не стоит называть переменные без какого-то смыслового значения, например, называть переменную буквой `a`
- Название переменной не может начинаться с цифры

```
# Название переменной начинается с цифры
1st_variable = 'some text'
```

Output:

```
File "<ipython-input-7-e01232b9e138>", line 1
    1st_variable = 'some text'
```

```
                ^
SyntaxError: invalid syntax
```

- Нельзя обращаться к несуществующей переменной

```
# Обращение к несуществующей переменной
_snake_case_variable
```

Output:

```
---------------------------------------------------------------------
NameError                                 Traceback (most recent c
<ipython-input-8-558a19c57801> in <module>
      1 # обращение к несуществующей переменной
----> 2 _snake_case_variable
NameError: name '_snake_case_variable' is not defined
```

Плохая практика

- Следует писать переменные в английской раскладке без использования кириллицы

```
# Русская "а"
а = 10
print(а)
```

Output:

```
10
```

```
# Латинская "a"
a
```

Output:

```
---------------------------------------------------------------------
NameError                                 Traceback (most recent c
<ipython-input-10-c844c2eb0517> in <module>
      1 # латинская "a"
----> 2 a
NameError: name 'a' is not defined
```

В Python тоже есть свой этикет, то есть правила форматирования кода. Он называется PEP8.

PEP8: https://pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html

> Типы данных в Python

В Python применяется динамическая типизация, иначе утиная.

Утиная типизация — если это выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, и есть утка. Это значит, что во время создания переменной ей сразу присваивается определенный тип данных.

Как узнать тип переменной

Для этого существует функция `type(<var>)`,

где `<var>` — название переменной.

Типы данных

В Python есть следующие встроенные типы данных:

1. `int` — целые числа
2. `float` — числа с плавающей точкой
3. `bool` — логический тип
4. `str` — строки
5. `list` — списки
6. `tuple` — кортежи

7. `set` — множества
8. `frozenset` — неизменяемые множества
9. `dict` — словари
10. `range` — диапазоны
11. `NoneType` — тип `None`

> Целые числа (int)

`int` представляет собой целое число. С ним можно проводить любые арифметические операции.

Примеры, где может применяться `int`:

- количество слов в словаре
- количество людей в комнате

Предположим, мы посчитали число людей в кабинете и опенспейсе, записали значения в переменные:

```
# Число людей в опенспейсе
openspace = 10

# Число людей в кабинете
cabinet = 3
```

Переменные имеют тип `int`

```
type(openspace)
```

Output:

```
<class 'int'>
```

Сложение

```
openspace + cabinet
```

Output:

```
13
```

Вычитание

```
openspace - cabinet
```

Output:

```
7
```

Умножение

```
openspace * cabinet
```

Output:

```
30
```

Деление

```
openspace / cabinet
```

Output:

```
3.3333333333333335
```

Целочисленное деление

```
openspace // cabinet
```

Output:

```
3
```

Остаток от деления

```
openspace % cabinet
```

Output:

```
1
```

Возведение в степень

```
cabinet ** 3
```

Output:

```
27
```

Сравнение чисел

```
# >, < — больше, меньше
# >=, <= — больше или равно, меньше или равно
# == — равно, не путать с присваиванием
# != — не равно

openspace > cabinet
```

Output:

результат: `bool`

```
True
```

Частая ошибка: путаница с `==` и `=`

- `==` — сравнение равенства
- `=` — присвоение значение

```
print(openspace == 5)
print(openspace)
```

Output:

```
10
```

```
openspace = 5
print(openspace)
```

Output:

```
5
```

> Числа с «плавающей запятой» (float)

`float` — числа с «плавающей запятой».

Работать с ними можно аналогично целым.

Примеры:

- стоимость товара в магазине (копейки как сотые рубля)

Рассмотрим тип `float` на примере цен:

```
good1 = 149.
good2 = 109.99
```

Видим, что это `float`

```
type(good1)
```

Output:

```
<class 'float'>
```

Сложение

```
good1 + good2
```

Output:

```
258.99
```

Вычитание

```
good1 - good2
```

Output:

Обратите внимание на результат:

```
39.010000000000005
```

Умножение

```
good1 * good2
```

Output:

```
16388.51
```

Деление

```
good1 / good2
```

Output:

```
1.3546686062369306
```

Целочисленное деление

```
good1 // good2
```

Output:

```
1.0
```

Остаток от деления

```
good1 % good2
```

Output:

```
39.010000000000005
```

Возведение в степень

```
good1 ** 2
```

Output:

```
22201.0
```

Сравнение чисел

```
good1 > good2
```

Output:

результат — `bool`

```
True
```

> Булевы значения (bool)

`bool` в основном используются в условных конструкциях (будут рассмотрены в следующем занятии).

Переменные данного типа принимают 2 значения:

- `True` — когда некоторое выражение истинно
- `False` — когда некоторое выражение ложно

Логические операторы:

- `not` — НЕ
- `or` — ИЛИ
- `and` — И

Примеры:

- болел ли человек гриппом
- видна ли кнопка на сайте

```
# человек болел
ill = True

# человек не болел
not_ill = False
```

Сравним переменные болел или не болел друг с другом

```
ill == not_ill
```

Output:

```
False
```

Таблицы истинности

Таблица истинности — таблица, описывающая логическую функцию.

```
print(f'Сложение (логическое ИЛИ) логических типов: {ill or not_il
print(f'Умножение (логическое И) логических типов: {ill and not_il
print(f'Отрицание (логическое НЕ) логических типов: {not ill}')
```

Output:

```
Сложение (логическое ИЛИ) логических типов: True
Умножение (логическое И) логических типов: False
Отрицание (логическое НЕ) логических типов: False
```

```
print(f'Сложение логических типов: {ill + not_ill}')
print(f'Умножение логических типов: {ill * not_ill}')
```

Output:

```
Сложение логических типов: 1
Умножение логических типов: 0
```

```
print(f'Числовая операция умножения с логическим типом: {ill * 3}'
print(f'Числовая операция сложения с логическим типом: {ill + 3}')
```

Output:

```
Числовая операция умножения с логическим типом: 3
Числовая операция сложения с логическим типом: 4
```

Note: с f-строками и функцией `print()` мы познакомимся чуть позже.

Дополнительно с типами данных можно ознакомиться тут.

> Списки (list), множества (set), кортежи (tuple)

list

Список — один из встроенных типов данных, который используется для хранения нескольких значений в одной переменной.

Свойства:

- Упорядоченные изменяемые коллекции объектов
- Могут хранить разные типы данных одновременно

Создание списка:

- При помощи `[ ]`: `l = ['s', 'p', ['isok'], 2]`
- Из итерируемого объекта: `list(iterable_object)`
- При помощи генератора списка (это изучим позже)

Пример использования:

- набор оценок студентов
- слова предложения

Для примера возьмем список `colors`

```
colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']
```

Обозначим `<list>` — конкретный объект типа `list`

Длина списка

`len(<list>)` — позволяет узнать длину списка

```
len(colors)
```

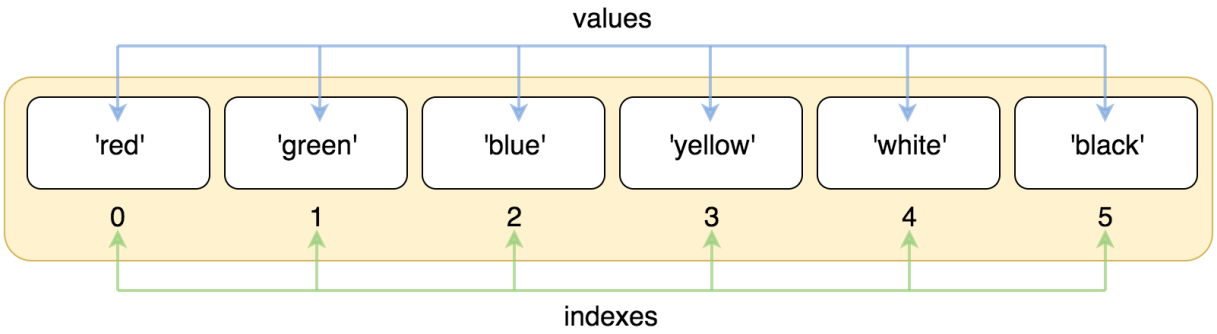
Output:

```
6
```

Списки поддерживают индексацию

`<list>[index]` — получим элемент списка на позиции `index`

В Python принят отсчет от `0`. Поэтому индексирование по `0` выдаст первый элемент списка, а индексирование по `1` — второй элемент списка.

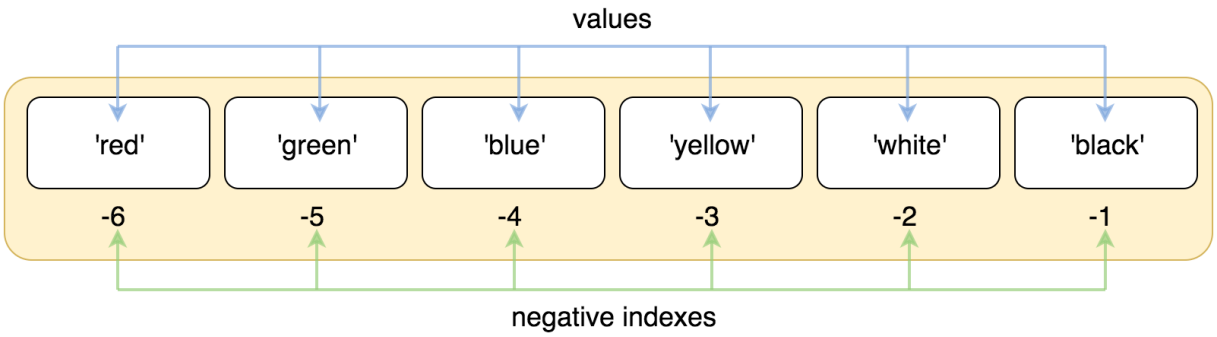



```
index = 1
print(f"Доступ к элементу, равному {colors[index]}, по индексу {in
```

Output:

```
Доступ к элементу, равному green, по индексу 1
```

Можно индексироваться с конца, то есть элемент с индексом -1 — последний элемент с конца.



```
index = -1
print(f"Доступ к элементу, равному {colors[index]}, по индексу {in
```

Output:

```
Доступ к элементу, равному black, по индексу -1
```

Если вызываемого индекса нет, то Python выдаст ошибку.

```
# Выход за границы - ошибка
index = 10
colors[index]
```

Output:

```
---------------------------------------------------------------------
IndexError                                Traceback (most recent c
<ipython-input-30-7a43aca826be> in <module>
      1 # выход за границы - ошибка
      2 index = 10
----> 3 colors[index]
IndexError: list index out of range
```

Списки поддерживают слайсы

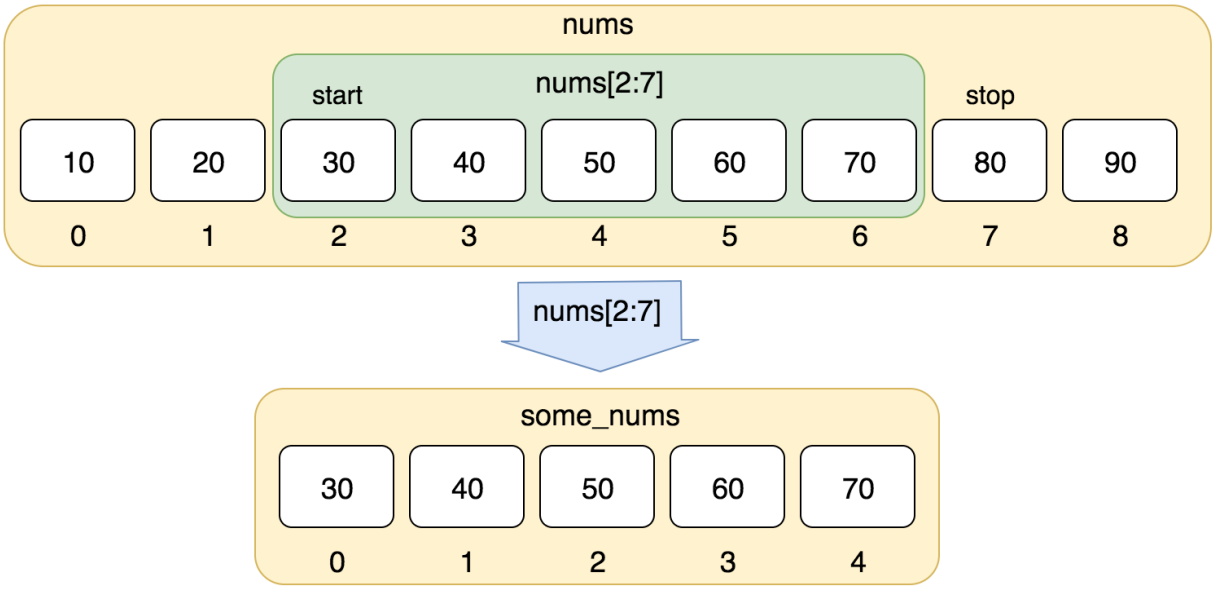
`<list>[start:stop:step]` — вырежем кусочек списка, начиная со `start`, заканчивая `stop` — 1, используя шаг `step`:

- `start` — начальный индекс. При отсутствии используется `0`
- `stop` — конечный индекс. При отсутствии будет длина — `1`. Важно отметить, что элемент с данным индексом не будет использован
- `step` — шаг. При отсутствии равен `1`

Давайте создадим список `age_list`

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

И возьмем слайс `[2:7]` от начала списка до 2 элемента.




```
age_list[2:7]
```

Output:

```
[30, 40, 50, 60, 70]
```

Методы списка

Note: что такое методы и чем они отличаются от функций изучим в следующих лекциях.

Добавление элемента в конец списка

`<list>.append(<последовательность>)` — добавляет в конец списка последовательность

```
age_list.append(1)
print(f"Добавление элемента в конец списка: {age_list}")
```

Output:

```
Добавление элемента в конец списка: [10, 20, 30, 40, 50, 60, 70, 8
```

```
building_class = ['a', 'b']
additional_building_class = ['a_plus', 'c']
building_class.append(additional_building_class)
print(f"Добавление списка в конец другого списка: {building_class}
```

Output:

```
Добавление списка в конец другого списка: ['a', 'b', ['a_plus', 'c
```

Добавление элементов из последовательности в конец списка

`<list>.extend(<последовательность>)` — добавляет в конец списка элементы из последовательности

```
age_list.extend([6, 7, 'a'])
print(f"Добавление элементов другого списка в конец: {age_lis
t}")
```

Output:

```
Добавление элементов другого списка в конец: [10, 20, 30, 40, 5
0, 60, 70, 80, 90, 1, 6, 7, 'a']
```

Вставка элемента на указанную позицию

`<list>.insert(index, element)` — добавляет элемент `element` на позицию `index`

```
age_list.insert(0, 2)
print(f"Вставка элемента на указанную позицию: {age_list}")
```

Output:

```
Вставка элемента на указанную позицию: [2, 10, 20, 30, 40, 50, 6
```

```
0, 70, 80, 90, 1, 6, 7, 'a']
```

Удаление элемента с заданным индексом

`<list>.pop(index)` — удаляет элемент с индексом `index` и возвращает его пользователю

```
popped = age_list.pop(4)
print(f"Удаление элемента с заданным индексом: {age_list}, poppe
d = {popped}")
```

Output:

```
Удаление элемента с заданным индексом: [2, 10, 20, 30, 50, 60, 7
0, 80, 90, 1, 6, 7, 'a'], popped = 40
```

Удаление элемента с заданным значением

`<list>.remove(element)` — удаляет заданный элемент `element`. Если таких элементов несколько, то удалит только первый

```
age_list.remove('a')
print(f"Удаление элемента с заданным значением: {age_list}")
```

Output:

```
Удаление элемента с заданным значением: [2, 10, 20, 30, 50, 60, 70
```

Работает только если в списке сравнимые типы данных (например, можно сравнить `int`, `float`, `bool`, но `str` сравнивать с `int` нельзя)

Максимальный и минимальный элемент

- `max(<list>)` — возвращает максимальный элемент в списке
- `min(<list>)` — возвращает минимальных элемент в списке

```
print(f"Максимальный элемент: {max(age_list)}")
```

```
print(f"Минимальный элемент: {min(age_list)}")
```

Output:

```
Максимальный элемент: 90
Минимальный элемент: 1
```

С другими функциями для работы со списками можно ознакомиться тут.

Set

Сеты (множества) аналогичны математическим множествам.

- В сете находятся только уникальные значения
- Сеты не поддерживают индексы — неупорядоченная структура
- Может хранить почти любые типы данных (только неизменяемые)

Создание:

- При помощи `{}`: `s = {'set', 's', True, 2}` . Создать пустое множество так нельзя (литерал принадлежит `dict`)
- Из итерируемого объекта: `set(iterable_object)`

Пример использования:

- обход графов
- общие товары с конкурентом

Часто, имея список, необходимо выделить из него только уникальные элементы — тут поможет `set`.

Создадим список `opinions` и переведем его в `set`, с помощью `set(opinions)`

`opinions = ['хорошо', 'хорошо', 'отлично', 'можно лучше', 'отлично', 'пас']`

```
unique_opinions = set(opinions)
```

Output:

```
unique_opinions
{'можно лучше', 'отлично', 'пас', 'хорошо'}
```

Основные функции

Для работы с сетами полезны следующие функции:

Добавление элемента

`<set>.add(element)` — добавляет элемент element

```
# Добавление нового элемента
client_group_avito.add('нет телефона')
client_group_avito
```

Output:

```
{89159001333, 89859999333, 89999999333, 'нет телефона'}
```

```
# Добавление элемента, который уже есть в сете
client_group_avito.add(8_985_999_93_33)
client_group_avito
```

Output:

```

{89159001333, 89859999333, 89999999333, 'нет телефона'}
```

Удаление элемента из сета

`<set>.remove(element)` — удаляет заданный элемент `element`

```
# Удаление элемента из сета
client_group_avito.remove('нет телефона')
client_group_avito
```

Output:

```
{89159001333, 89859999333, 89999999333}
```

Объединение двух сетов

`<set1>.union(<set2>)` — возвращает объединение 2 сетов

```
client_group_cian = {8_985_999_93_33, 8_915_900_13_33, 8_800_555_3
client_group_avito.union(client_group_cian)
```

Output:

```
{88005553535, 89159001333, 89859999333, 89999999333}
```

Пересечение сетов

`<set1>.intersection(<set2>)` — возвращает пересечение 2 сетов

```
client_group_avito.intersection(client_group_cian)
```

Output:

```
{89159001333, 89859999333}
```

```
# Состояние не изменилось
client_group_avito
```

Output:

```
{89159001333, 89859999333, 89999999333}
```

Разность сетов

`<set1>.difference(<set2>)` — возвращает разность 2 сетов

Другие функции:

`len(<set>)` — размер сета

`max(<set>)` — возвращает максимальный элемент в сете

`min(<set>)` — возвращает минимальный элемент в сете

```
print(f"Размер сета: {len(client_group_avito)}")
print(f"Максимальный элемент: {max(client_group_avito)}")
print(f"Минимальный элемент: {min(client_group_avito)}")
```

Output:

```
Размер сета: 3
Максимальный элемент: 89999999333
Минимальный элемент: 89159001333
```

Проверка вхождения в сет

`element in <set>` — проверка вхождения в `set`

```
element = 8_800_555_35_35
print(f'{element} in {client_group_avito}: {element in client_grou
```

Output:

```
88005553535 in {89159001333, 89859999333, 89999999333}: False
```

С другими функциями для работы с сетами можно ознакомиться тут.

Tuple

Кортеж (тапл, тьюпл) очень напоминает списки, с той разницей, что в тапл нельзя добавить элемент или удалить его после создания объекта (неизменяемый тип данных). Но к элементам можно обращаться по индексу.

Зачем нужен кортеж?

- более быстрый аналог списка
- защищает хранимые данные от непреднамеренных изменений
- можно использовать как ключ в словаре

Создание:

- При помощи `()`: `t = ('t', 'u', ['ple'], 2)`
- Из итерируемого объекта: `tuple(iterable_object)`

Сохраним в переменную `server_response` тапл

```
server_response = (36.6, 39.0, 37.1, 36.8)
```

Доступ к элементу по индексу

```
server_response[1]
```

Output:

```
39.0
```

Слайс от начала тапла до 2го элемента

```
server_response[:2]
```

Output:

```
(36.6, 39.0)
```

Другие функции:

`len(<tuple>)` — размер тапла

`max(<tuple>)` — возвращает максимальный элемент в тапле

`min(<tuple>)` — возвращает минимальный элемент в тапле

```
print(f"Длина тапла: {len(server_response)}")
print(f"Максимальный элемент: {max(server_response)}")
print(f"Минимальный элемент: {min(server_response)}")
```

Output:

```
Длина тапла: 4
Максимальный элемент: 39.0
Минимальный элемент: 36.6
```

Количество элементов

`<tuple>.count(element)` — возвращает кол-во элементов, равных element

```
print(f"Кол-во элементов, равных заданному: {server_response.count
```

Output:

```
Кол-во элементов, равных заданному: 0
```

Индекс первого вхождения переданного элемента

`<tuple>.index(element)` — возвращает индекс первого вхождения element в `<tuple>`

```
print(f"Индекс первого вхождения переданного элемента: {server_res
```

Output:

```
Индекс первого вхождения переданного элемента: 0
```

С функциями для работы с таплами можно ознакомиться тут.

> Строки (string)

Данный тип может представлять собой любой текст: хоть один символ, хоть весь текст какой-то книги.

Интерактивное взаимодействие с пользователем через `input()` — строка.

Строки в Python можно задавать как при помощи двойных, так и при помощи одинарных кавычек, а также вызова `str()`.

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

```
print('Hello world')
```

Output:

```
Hello world
Hello world
```

Если нужно создать строку, которая занимает несколько строк, то необходимо в начале и в конце строки поставить 3 кавычки

```
print(
"""Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
)
```

Output:

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.
```

f-строки

В Python существует форматирование строк (в данном случае f-строки).

f-строки в Python позволяют встраивать значения переменных и выражений непосредственно в строку, делая код более читаемым и удобным.

Для создания f-строки необходимо перед строкой поставить префикс "f" или "F", а затем внутри строки использовать фигурные скобки {} для вставки переменных или выражений.

```
name = "Ivan"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

Output:

```
My name is Ivan and I am 30 years old.
```

Подробнее с `f`-строками можно ознакомиться тут

Индексация и слайсы

Также строки поддерживают индексацию и слайсы. Это позволяет получить конкретный символ или набор символов.

Слайс (срезы) — выражение вида `string[start:end:step]`, где:

1. `start` — начальный индекс. При отсутствии используется начало строки
2. `end` — конечный индекс. При отсутствии будет конец строки. Важно отметить, что символ с данным индексом не будет использован
3. `step` — шаг. При отсутствии равен 1

```
print(f"Слайс строки с 6 по 9 элементы: {s[6:9]}")
print(f"Слайс от начала строки до 9 элемента {s[:9]}")
print(f"Слайс от 6 элемента до конца строки: {s[6:]}")
print(f"Слайс с каждым четным символом: {s[::2]}"
```

Output:

```
Слайс строки с 6 по 9 элементы: wor
Слайс от начала строки до 9 элемента Hello wor
Слайс от 6 элемента до конца строки: world
Слайс с каждым четным символом: Hlowrd
```

Доступ к элементу по индексу

```
print(f"Доступ к элементу по индексу: {s[0]}")
```

Output:

```
Доступ к элементу по индексу: H
```

Доступ к последнему символу

```
print(f"Доступ к последнему символу: {s[-1]}")
```

Output:

```
Доступ к последнему символу: d
```

Длина строки

`len(<str>)` — найти длину строки

```
print(f'Длина строки {len(s)}')
```

Output:

```
Длина строки 11
```

К строкам можно применять некоторые операции, применимые к числам.

При сравнении строк принимается во внимание символы и их регистр:

- цифровой символ условно меньше, чем любой алфавитный символ
- алфавитный символ в верхнем регистре условно меньше, чем алфавитные символы в нижнем регистре

```
str1 = 'a'
str2 = 'c'
str3 = '10'
print(f"{str1} == {str2}: {str1 == str2}")
print(f"{str1} <= {str2}: {str1 <= str2}")
print(f"{str1} > {str3}: {str1 > str3}")
```

Output:

```
a == c: False
a <= c: True
```

```
a > 10: True
```

```
print(f"{str1} + {str2}: {str1 + str2}")

multiplier = 10
print(f"{str1} * {multiplier}: {str1 * multiplier}")
```

Output:

```
a + c: ac
a * 10: aaaaaaaaaa
```

Основные методы

```
s = " _Hello World_"
print("Исходная строка:", s)
```

Output:

```
Исходная строка:' _Hello World_'
```

Обрезка строки по символу

- `<str>.strip()` — позволяет удалить определенные символы в начале и в конце строки
- `<str>.lstrip()` — удаляет символы только в начале строки
- `<str>.rstrip()` — удаляет символы только в конце строки

```
print(f"Обрезка строки по символу '_': '{s.strip('_')}'")
print(f"Обрезка строки по символу ' ': '{s.lstrip()}'")
print(f"Обрезка строки справа по символу '_': '{s.rstrip('_')}'")
```

Output:

```
Обрезка строки по символу '_': ' _Hello World'
Обрезка строки по символу ' ': '_Hello World_'
Обрезка строки справа по символу '_': ' _Hello World'
```

Замена символов в строке

`<str>.replace(char_1, char_2)` — позволяет заменить все символы `char_1` в строке на `char_2`

```
print(f"Замена сиволов в строке {s.replace('o', 'O')}")
```

Output:

```
Замена сиволов в строке  _HellO WOrld_
```

Разделение строки по пробелам

`<str>.split(char)` — позволяет разделить строку на список подстрок, в качестве разделителя будет использован `char`. Если `char` не задан, то строка разбивается по пробелу.

```
print(f"Разделение строки по пробелу {s.split()}")
```

Output:

```
Разделение строки по пробелу [' _Hello', 'World_']
```

Объединение списка ['Element_1', 'Element_2'] в строку

`<str>.join(<list of strs>)` — объединяет список элементов в строку. Между каждым элементом будет стоять строка, для которой вызывалась данная функция.

```
print(f"Объеденение списка в строку: {', '.join(['Element_1', 'Ele
```

Output:

```
Объеденение списка в строку: Element_1, Element_2
```

Подробнее про строковые методы тут.

> Словари (dict), range, None

dict

Представляет собой набор пар `ключ:значение.`

- `ключ` — только неизменяемые типы данных (например: `int`, `float`, `decimal`, `complex`, `bool`, `str`, `tuple`, `range`, `frozenset`, `bytes`)
- `значения` — любой тип данных

Аналог ключа в `list` — индекс

Создание

- При помощи `{}`: `d1 = {"Russia": "Moscow", "USA": "Washington"}`
- При помощи функции: `dict(Ivan="manager", Mark="worker")`
- При помощи генератора (изучим позже)

Примеры использования:

- ключ — ФИО, значение — возраст
- ключ — дата, значение — список уникальных покупателей

Создадим словарь `user_phones`

```
user_phones = {
    "user1": 89859999333,
    "user2": 89999999333,
    "user3": 'нет телефона',
    "user4": None,
}
```

Output:

```
# Исходный словарь
user_phones
{'user1': 89859999333,
 'user2': 89999999333,
 'user3': 'нет телефона',
 'user4': None}
```

Функции для работы со словарями:

Обновленный словарь

`<dict>.update(dict)` — добавляет в словарь новые пары `ключ:значение`. Если один из ключей присутствует в словаре, то значение перезаписывается

```
user_phones.update({"user3": 8_486_250_00_00,
                    "user5": [8_900_000_12_12, 3_345_233_11_94]})
```

Output:

```
# Обновленный словарь
user_phones
{'user1': 89859999333,
 'user2': 89999999333,
 'user3': 84862500000,
 'user4': None,
 'user5': [89000001212, 33452331194]}
```

Элементы словаря

`<dict>.items()` — возвращает список таплов вида (ключ, значение)

```
# Элементы словаря
user_phones.items()
```

Output:

```
dict_items([('user1', 89859999333), ('user2', 89999999333), ('us
er3', 84862500000),
('user4', None), ('user5', [89000001212, 33452331194])])
```

- `<dict>.keys()` — возвращает список ключей
- `<dict>.values()` — возвращает список значений

```
# Список ключей
user_phones.keys()
# Список значений
user_phones.values()
```

Output:

```
dict_keys(['user1', 'user2', 'user3', 'user4', 'user5'])
dict_values([89859999333, 89999999333, 84862500000, None, [8900000
```

К элементам словаря можно обращаться по ключу. Например, мы хотим обратиться по ключу `user1` ко всем элементам словаря `user_phones`. Для этого достаточно написать `user_phones['user1']`:

```
user_phones['user1']
```

Output:

```
89859999333
```

Обратите внимание на работу `f-строк` и словарей.

```
# Создаем словарь
my_dict = {'website': 'example.com', 'course_name': 'python_cour
se'}

# Используем f-строку с экранированием кавычек для выделения наз
ваний ключей словаря
formatted_string = f"{my_dict['website']}/{my_dict['course_nam
```

```
e']}"
print(formatted_string)
```

Output:

```
example.com/python_course
```

В данном случае мы использовали как двойные кавычки `""` для выделения f-строки в целом, так и одинарные `''` для выделения названий ключей словаря. Это необходимо для того, чтобы правильно интерпретировать строку и избежать ошибок при использовании `f-строк` в Python.

Подробнее про словари тут.

range()

Представляет собой набор целых чисел.

`range([start], stop, [step])` — задает набор целых чисел от `start` до `stop` с шагом `step`. По умолчанию `start` равен `0` и step равен `1` .

Зачем нужен:

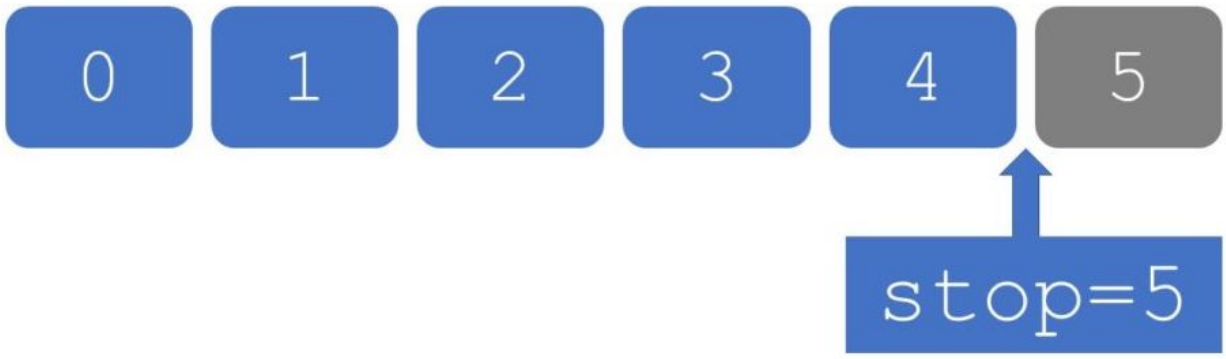
- полезно для создания прогрессий
- очень часто используется в циклах (изучим в следующей лекции)

Указан только один аргумент — stop

```
marks = range(5)
print(marks, list(marks))
```

Output:

```
range(0, 5) [0, 1, 2, 3, 4]
```



Указаны два аргумента — start и stop

```
marks = range(1, 5)
print(marks, list(marks))
```

Output:

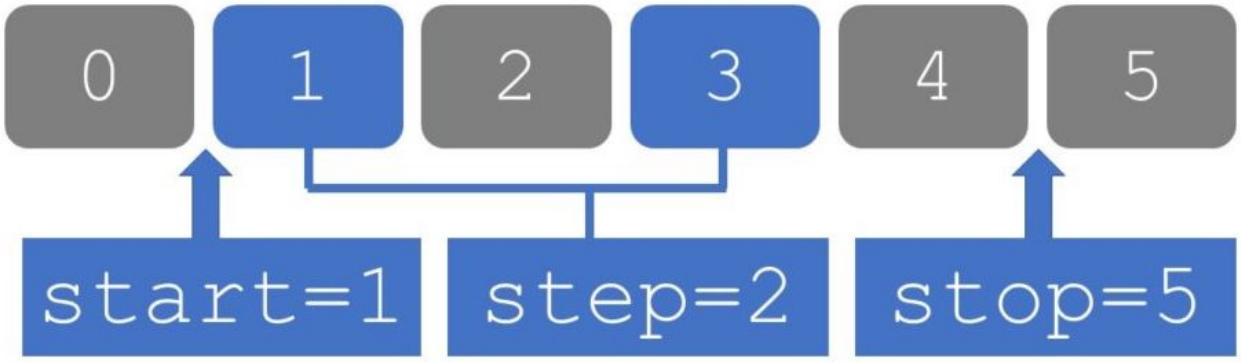
```
range(1, 5) [1, 2, 3, 4]
```

Указаны три аргумента — start, stop и step

```
odd = range(1, 5, 2)
print(odd, list(odd))
```

Output:

```
range(1, 5, 2) [1, 3]
```



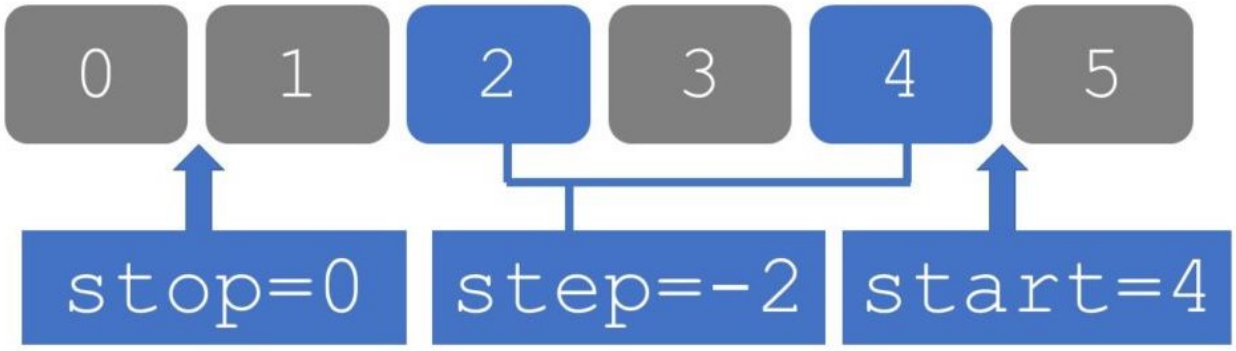
Указаны три аргумента — start, stop и отрицательный step

- порядок обратный
- start > stop

```
even = range(4, 0, -2)
print(even, list(even))
```

Output:

```
range(4, 0, -2) [4, 2]
```



None

В Python тип данных `None` представляет собой специальное значение, которое используется для обозначения отсутствия значения или отсутствия результата.

`None` является единственным значением типа данных `NoneType`.

```
type(None)
```

Output:

```
NoneType
```

> Немного о работе с типами данных

Некоторые полезные функции при работе с типами данных:

- `type(variable)` — возвращает тип данных для variable
- явная конвертация — `<data_type>(variable)` — переводит тип данных `variable` к типу `<data_type>` (например: `int(variable)` — попытка привести `variable` к типу `int`)
- неявная конвертация происходит при операциях с разными типами

```
openspace = 10
print(f"Тип данных для openspace == {openspace}: {type(openspace)}
print(f"Тип данных для float(openspace) == {float(openspace)}:{typ
print(f"Тип данных для str(openspace) == {str(openspace)}: {type(s
```

Output:

```
Тип данных для openspace == 10: <class 'int'>
Тип данных для float(openspace) == 10.0: <class 'float'>
Тип данных для str(openspace) == 10: <class 'str'>
```

int()	string, floating point to integer
float()	string, integer to floating point number
str()	integer, float, list, tuple, dictionary to string
list()	string, tuple, dictionary to list
tuple()	string, list to tuple

```
item_price = 149.99
print(f"Тип данных для item_price == {item_price}: {type(item_pric
# Обратите внимание
print(f"Тип данных для int(item_price) == {int(item_price)}: {type
```

Output:

```
Тип данных для item_price == 149.99: <class 'float'>
Тип данных для int(item_price) == 149: <class 'int'>
```

Важно отметить, что не все типы могут быть сконвертированы друг в друга.

Видим, что строка не может переконвертироваться в int.

```
username = 'Alexandra'
print(f"Тип данных для username: {type(username)}")
```

Output:

```
Тип данных для username: <class 'str'>
```

```
# Конвертация в int
int(username)
```

Output:

```
---------------------------------------------------------------------
ValueError                                Traceback (most recent c
<ipython-input-88-2a63fdda5cde> in <module>
----> 1 int(username)
ValueError: invalid literal for int() with base 10: 'Alexandra'
```

Другие примеры

```
user_input = '-0.60'
print(f"Тип данных для user_input == {user_input}: {type(user_inpu
```

```
print(f"Тип данных для float(user_input) == {float(user_input)}: {
```

Output:

```
Тип данных для user_input == -0.60: <class 'str'>
Тип данных для float(user_input) == -0.6: <class 'float'>
```

```
item_supply = [10, 2, 0, 10]
print(f"Тип данных для item_supply == {item_supply}: {type(item_su
print(f"Тип данных для set(item_supply) == {set(item_supply)}: {ty
```

Output:

```
Тип данных для item_supply == [10, 2, 0, 10]: <class 'list'>
Тип данных для set(item_supply) == {0, 10, 2}: <class 'set'>
```

```
country_capital = (('Россия', 'Москва'),
                   ('Польша', 'Варшава'))
print(f"Тип данных для country_capital == {country_capital}: {ty
pe(country_capital)}")
print(f"Тип данных для dict(country_capital) == {dict(country_ca
pital)}:",
      f"{type(dict(country_capital))}")
```

Output:

```
Тип данных для country_capital == (('Россия', 'Москва'), ('Польш
а', 'Варшава')):
<class 'tuple'>
Тип данных для dict(country_capital) == {'Россия': 'Москва', 'По
льша': 'Варшава'}:
<class 'dict'>
```

> Дополнительные материалы

1. Больше информации о математических операциях в питоне
2. Статья про f-строки
3. Подробнее об операторах
4. Индексация и слайсы
5. Правила оформления кода
6. Функция range()
7. Полезный сайт про Python