BaseFunctionsPy_UA

Вступ

У цьому гайді ви зможете отримати базові знання по Python. Причиною написання гайда була важкість і заплутаність яку я помітив в інших гайдах. Безумовно я не експерт в Python, але при цьому маю якийсь досвіт та думаю, що можу допомогти цими знаннями іншим :)

Корисні посилання

Так як цей гайд достатньо суб'єктивний я не впевнений, що захвачу усі теми і не впевнений, що у вас не з'являться питання після читання, через що в цьому блокі прикріплено гарні посилання, які можна використовувати в таких випадках.

В першу чергу ви можете написати мені на пошту і я спробую вам допомогти:

boyarkin.gleb@gmail.com

source.boar@gmail.com

Також пошти продубльовані у мене в профілі.

Окрім мене ви можете використати:

boyarkin.gleb@gmail.com

source.boar@gmail.com

Так же почты подублированы у меня в профиле.

Кроме меня вы можете воспользоваться:

https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference

Ці джерела гарантую, що є перевіренними, сам по ним навчався, але, як на мене, багато речей там розписані надто складними словами.

Коротко про JavaScript

JavaScript (JS) - досить молода і відома динамічна мова програмування. Основною галуззю застосування є насамперед Web розробка, а також розробка додатків.

Синтаксис

Синтаксис JS здебільшого є C-подібним, але зі своїми дрібними відмінностями, з якими ми зіштовхуватимемося по ходу.

Якщо проходитись за основними правилами синтаксису, то:

- `;` – символ закінчення рядка
- `{}` – сиволи для відкриття та закриття блоків коду
- `camelCase` – стиль іменування змінний та функцій

- `PascalCase` – стиль іменування класів
- `//` – символ для коментаря одного рядка
- `/* */` – символи для коментування блоку коду

Дрібна важлива замітка

`consoe.log(x);` - функція для виведення якогось `x` в консоль. Конкретніше про це буде далі, але це потрібно для розуміння найближчої ділянки коду

Змінні

Як я думаю вам відомо, чи стане відомо зараз, за фактом вся мова програмування тримається на змінних та операціях над ними.

У JS змінна - це покажчик на область або осередок пам'яті, в якій зберігається якесь зниження. Факт про пам'ять поки що можна просто покласти в голову, до нього ми дійдемо потім.

Для оголошення змінних використовуються ключові слова `var`, `let` та `const`.

Приклад оголошення:

```
var a1 = "Hello";
let a2 = 1;
const A3 = true;
```

При іменуванні змінних, як було зазначено вище, використовується стиль camelCase.

Що ж до нього входить?

camelCase:

- перша літера – нижнього регістру
- кожна перша літера наступного слова – верхнього регістру

Так само якщо про JS то в ньому назва змінної може починати з символу `_` або літери будь-якого регістру, наступні символи вже можуть бути такі ж плюс цифри (0-9). Також варто уточнити, що JS чутливий до регістру, тобто в ньому `a` і `A` це різні імена.

Різниця між `var`, `let` та `const`

- `var` – спосіб оголошення змінної з контекстною областю видимості, що не вимагає значення при оголошенні, за замовчуванням `undefined`, змінна може бути переоголошена. Цей спосіб оголошення вважається застарілим, і рекомендується замість нього використовувати `let`
- `let` – спосіб оголошення змінної з блоковою областю видимості, що не вимагає значення при оголошенні, за замовчуванням `undefined`, змінна не може бути переоголошена.
- `const` – спосіб оголошення змінної з блоковою областю видимості, але при цьому доступ надалі до змінної після оголошення тільки для читання, з цього виходить, що значення при оголошенні є

обов'язковим. Так само константі змінні прийнято називати літерами верхнього регістру, але це не є обов'язковим.

- `Без ключового слова` – спосіб оголошення змінної з глоабльною областю видимості

А тепер простими словами

Область видимості — ділянка програми, на якій ваша змінна, функція, клас будуть видні іншим об'єктам програми.

Глобальна область видимості — областю видимості є вся програма

Блочна область видимості – область видимості в межах блоку програми

Блок програми – функція, клас, вся програма

Контекстна область видимості — область видимості залежно від контексту оголошення: при оголошенні в функції області видимості буде ця функція, при оголошенні у всій програмі область видимості буде вся програма ітд.

Приклади

```
var a1 = 1;
let a2 = 2;
const A3 = 3;
a4 = 4;

function f(){
    var f1 = 1;
    let f2 = 2;
    const F3 = 3;
    f4 = 4;
}

f();

function g(){
    console.log(a1); // виведе
    console.log(a2); // виведе
    console.log(A3); // виведе
    console.log(a4); // виведе
    console.log(f1); // не виведе
    console.log(f2); // не виведе
    console.log(F3); // не виведе
    console.log(f4); // виведе
}

g();

function h(){
    var a1 = 0; // спрацює
    let a2 = 0; // спрацює "але"
    const A3 = 0; // спрацює "але"
    a4 = 0; // спрацює
```

```
    f4 = 0; // спрацює
}

h();

var a1 = 0; // спрацює
let a2 = 0; // не спрацює
const A3 = 0; // не спрацює
a4 = 0; // спрацює
```

А тепер розберемо що відбувається:

1. Змінні `a1`, `a2`, `A3`, `a4` оголошені на рівні програми, а `f1`, `f2`, `F3`, `f4` на рівні функції.
2. Що буде з `var`:
    1. Через контекстну типізацію `a1` стане глобальною, а `f1` локальною змінними, через що у функції `g` `a1` буде видно, а `f1` немає
    2. У функції `h` або в самій програмі змінну `a1` можна буде оголосити заново
3. Що буде з `let`:
    1. Через блокову типізацію `a2` буде видно всій програмі, а `f2` тільки функції в якій оголошено, через що у функції `g` `a2` буде видно, а `f2` немає
    2. У функції `h` змінну `a2` можна буде оголосити заново, "але" тепер `a2` для всього блоку програми буде новою і стара не буде доступна
    3. На рівні блоку, на якому була задана `a2`, переоголосити її вже не можна буде
4. Що буде з `const`:
    1. Через блокову типізацію `A3` буде видно всій програмі, а `F3` тільки функції в якій оголошено, через що у функції `g` `A3` буде видно, а `F3` немає
    2. У функції `h` змінну `A3` можна буде оголосити заново, "але" тепер `A3` для всього блоку програми буде новою і стара не буде доступна
    3. На рівні блоку, на якому була задана `A3`, переоголосити її вже не можна буде
    4. Змінити значення жодної із змінних цього типу буде неможливо
5. Що буде з `без ключового слова`:
    1. Обидві змінні стануть глобальними через глобальну область видимості
    2. Переоголошення не відрізняється від перепризначення так що технічно її можна буде змінювати як завгодно в будь-якому блоці програми.

Чому `var` вважається застарілою?

Як ми побачили вище: змінні оголошені через `var` можуть переоголошуватися, через що можуть виникати маси помилок і нестикування в коді. Щоб вирішити цю проблему був створений `let` який у свою чергу на

такі проблеми не страждає.

Типи даних

У JS є сім примітивних нібито даних.

** Тип даних ** — типи того, якого типу значення може зберігати змінна.

Примітиви (примітивні типи даних) — типи даних, які не є об'єктами і не мають методів.

Простими словами: найпростіші системні типи даних.

Примітиви в JS:

- `Boolean` – логічний тип даних, що зберігає 2 види значень: `true` або `false`
- `Number` – числовий тип даних, зберігає числа в діапазоні приблизно від *-2*10^307* до *2*10^307*, у разі передачі значення менше буде записуватись `-Infinity`, а більше – `Infinity`
- `String` – малий тип даних, що зберігає рядок тексту майже необмеженого розміру
- `null` – ключове слово, яке означає що нульове або "порожнє" значення
- `undefined` – ключове слово, яке означає, що змінна не зберігає жодного значення, за замовчуванням надається всім змінних при оголошенні без значення, за винятком `const`

Типізація

Типізація JS – динамічна, слабка, неявна.

А тепер спробуємо зрозуміти:

- Динамічна – означає, що змінна не прив'язана до єдиного типу при оголошенні, змінна може зберігати різні типи під час використання програми
- Слабка – означає що з виконанні дій зі змінною одного типу дій іншого типу нічого очікувати викликати помилку. * Відомий приклад *: `1 + "2" == "12"`, тобто що число спробували підсумувати з рядком, але це не викликало помилку і спрацювало
- Неявна - значить, що при оголошенні не потрібно "явно" вказувати тип змінної. *Наприклад як це в C++ з явною типізацією: * `string a = "a"`, коли в JS вистачить `let a = "a"`, без зазначення типу

Стаття, де ця тема розкрита глибше: https://tproger.ru/explain/tipizacija-jazykov-programmirovanija-razbiraemsja-v-osnovah/

Що ж нам дають ці пізнання?

З типізації JS ми можемо вилучити ті факти, що:

- Нам не потрібно стежити за типами на рівні програми
- Нам не потрібно оголошувати змінну вказуючи її тип
- Ми можемо для однієї змінної надавати значення різних типів

Розглянемо саме різні типи

Boolean

Це логічний тип даних, який використовується в місцях, де потрібно в змінній зберігати 2 варіанти значення: ТАК (`true`) або НІ (`false`)

Оголошення змінних цього типу можуть виглядати так:

```
let t = true;
let f = false;
```

Використовуються вони, в основному, в умовних конструкціях, які ми розглянемо пізніше.

Number

Числовий тип даних. Може зберігати різні числа: позитивні, негативні, нуль, цілі, нецілі...

Оголошення змінних цього типу можуть виглядати так:

```
let n = 2;
let p = 3.14;
let m = -4;
let o = 0;
```

Також для перетворення змінних іншого типу в `Number` можна використовувати функції `ParseInt(string)` , `ParseFloat(string)` або просто створення об'єкта `Number()`

String

Стічний тип даних зберігає в собі текст.

Оголошення змінних цього типу можуть виглядати так:

```
let txt1 = "Hello";
let txt2 = 'World';
let txt3 = `!`;
```

Різниці між лапками немає (за винятком ``, але про це не зараз), головне щоб вони відкривали і закривали рядок одночасно.

Також для перетворення змінних іншого типу в `String` можна використовувати функції `String()` або `.toString()`

Також для рядків існує дуже велика кількість методів, перелік яких та інструкції до них можна знайти за посиланням: https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String

Оператори

Оператори присвоєння

`=` – присвоєння

`+=` – присвоєння із додаванням

`-=` – присвоєння з відніманням

`*=` – присвоєння з множенням

`/=` – присвоєння з розподілом

`%=` – присвоєння з поділом із залишком

`**=` – присвоєння зі зведенням у ступінь

`++` – інкремент, теж саме що `+= 1`

`--` – декремент, теж саме що `-= 1`

`<<=` – присвоєння з лівим зрушенням

`>>=` – присвоєння з правим зрушенням

`>>>=` – присвоєння з беззнаковим зрушенням праворуч

`&=` – присвоєння з побітовим `І`

`|=` – присвоєння з побітовим `АБО`

`^=` – присвоєння з побітовим виключаючим `АБО`

Логічні оператори

`==` – рівність

`===` – рівність з перевіркою типів

`!=` – нерівність

`!==` – нерівність із перевіркою типів

`>` – більше

`>=` – більше чи одно

`<` – менше

`<=` – менше або одно

`&&` – логічне `І`, тобто обидві умови в конструкції `(умова 1)&&(умова 2)` повинні бути `true`

`||` – логічне `АБО`, тобто хоча б одна умова в конструкції `(умова 1)||(умова 2)` треба бути `true`

Арифметичні оператори

`+` – плюс

`-` – мінус

`*` – множення

`/` – розподіл

`%` – розподіл із залишком

`**` – зведення в ступінь

Побітові оператори

`&` – побитове `І`

`|` – побітове `АБО`

`^` – побітове що виключає `АБО`

`~` – побітове `НЕ`

`<<` – зрушення вліво

`>>` – зсув праворуч

`>>>` – зсув праворуч із заповненням нулями

Умовні конструкції

Змінні це звичайно добре, але ж треба якось з ними працювати і ось умовні конструкції це один із способів прописування логіки програми.

if...else

Найвідоміша і найчастіше використовувана конструкція.

У `if` передається умова, якщо умова істинна то виконується те, що в блоці `if` в іншому випадку те, що в блоці `else`

Синтаксис виглядає приблизно так:

```
if(/*умова: або логічне вираження чи змінна типу Boolean*/){
    // дія якщо умова == true
}
else{
    // дія якщо умова == false
}
```

Так же в эту конструкцию можно добавить `else if`

Пример применения:

```
if(/*умова 1: або логічне вираження чи змінна типу Boolean*/){
    // дія якщо умова 1 == true
}
else if(/*умова 2: або логічне вираження чи змінна типу Boolean*/){
    // дія якщо умова 1 == false, но условие 2 == true
}
...
else{
```

```
    // дія якщо умова 1 == false и действие 2 == false
}
```

Таких конструкцій `else if` можна додавати будь-яку кількість разів

switch..case...default

Також умовна конструкція яка вкорочує код який можна написати через `if...else if...else`

`switch...case...default` відрізняється тим, що ми в switch передаємо якусь змінну, а потім конструкція порівнює цю змінну з кожним значенням `case` і якщо ніякого збігу з `case` не знайшлося виконуватись блок `default`

Синтаксис виглядає приблизно так:

```
switch(/*змінна яку порівнюємо*/){
     case /*значення 1*/:
          // дія якщо збіглася значення 1
          break;
     case /*значення 2*/:
          // дія якщо збіглася значення 2
          break;
     ...
     default:
          // дія якщо жодне значення не співпало
          break;
}
```

`break` є обов'язковою командою в кінці блоків, щоб порівняння змінних не пішло далі

try...catch...finally

Умовний блок, за якого умовою виконання буде наявність або відсутність помилки в блоці коду `try`

Синтаксис виглядає приблизно так:

```
try{
     // блок коду, який відтворюватиме на пошук помилки
}
catch(/*об'єкт помилки типу Error*/){
     // дія якщо помилка
}
finally{
     // дію після виконання блоку, яке не відтворюватиме на пошук помилки
}
```

Докладніше про об'єкти Error:
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Error

Цикли

while

Цикл при якому дія виконується доки умова істинна

Синтаксис виглядає приблизно так:

```
while(/*умова: або логічний вираз або змінна типу Boolean*/){
    // дія якщо умова true
}
```

do...while

Можна сказати різновид циклу `while`, тільки в `while` спочатку перевіряється умова, а потім дію, а в `do...while` навпаки - спочатку дію, а потім перевірка

Синтаксис виглядає приблизно так:

```
do{
    // дія якщо умова true
} while(/*умова: або логічний вираз або змінна типу Boolean*/)
```

for

Вид циклу з дією, що виконується до початку обходу, умовою виконання та дією, яка виконується в кінці кожного проходу

Синтаксис виглядає приблизно так:

```
for(/*дія на початку*/; /*умова виконання*/; /*дія наприкінці кожного проходу*/){
    // дія поки що умова true
}
```

Ключові слова `break` та `continue`

`break` використовується для дострокового виходу з циклу, тобто завершення роботи циклу незалежно від умови

Приклад використання:

```
for(let i = 0; i < 10; i ++) {
    if(i % 3 == 0){ // якщо число ділиться на 3 без залишку
        break;
    }
    consoe.log(i);
}
```

Тобто цикл виконувався доки не наткнувся на перше число, яке ділиться на 3

`continue` використовується для дострокового припинення проходу циклу і переходу на наступний прохід, тобто використовується коли вам треба, щоб все після його виклику не виконувалося в цьому проході, а виконувалося відразу зробляє прохід

```
for(let i = 0; i < 10; i ++) {
    if(i % 3 == 0){ // якщо число ділиться на 3 без залишку
        continue;
    }
    consoe.log(i);
}
```

Тобто для чисел які ділитися на 3 не виконувався `console.log()`, а одразу виконувався наступний прогін циклу

Практичні завдання на даних знаннях

Для кращого закріплення пропоную виконати кілька простих завдань, які перевірять, наскільки ви зрозуміли цей рівень матеріалу, всі знання, які можуть бути потрібні для виконання завдань, є вище.

Всі розв'язки задач будуть у папці "Завдання" і будуть пронумеровані, але не рекомендую вам ними користуватися доки у вас самих не вийде

1. Допишіть на початок коду такі оголошення змінних, щоб інший код запрацював як написано в коментарі. Не всі місця для написання коду мають бути заповнені

   ```
   // запишіть ваші оголошення сюди
   function func1(){
       // або сюди
   }

   function func2(){
       try{
           A1 = 0;
       }
       catch(e){
           console.log("Не можна змінювати змінну A1");
       }
       console.log(a2);
   }

   func1();
   func2();
   ```

   (Мають відбутися всі виводи в консоль)

2. Напишіть цикл який виведе в консоль всі числа від 1 до 100 включно які ділитися на 5 або на 7 (`n % /*число*/ == 0` – умова подільності `n` на якесь число)

3. Напишіть блок коду який буде у разі виконання дії `throw ""` виводилася фраза "Відбулася помилка" і після цього виводилася фраза "Помилки оброблені"

Фукнції

Функції – ще один важливий інструмент із програмування. Якщо простими словами це об'єднання якоїсь ділянки коду для його подальшого виклику. У JS функції це об'єкт і з ними можна працювати як з об'єктом,

тобто записувати в змінні, передавати аргумент, писати методи і так далі.

Кожна функція у будь-якій мові програмування має:

- Ім'я
- Список аргументів (параметрів) – змінні, значення яких задається під час виклику функції. Оголошуються вони в `()` через кому. Так само при виклику їх значення вказуються при виклику так само в `()`. У блоці коду нашої функції вони відіграють роль звичайних змінних з блоковою областю видимості, тільки не потребують оголошення, тому що оголошуються при виклику функції
- Блок коду, який буде виконуватися при викликі. Записується в `{}`

Синтаксис виглядає приблизно так:

```
function /*ім'я функції*/(/*аргумент1*/, /*аргумент2*/.../*аргументN*/){
     // блок коду, що виконується
}
```

`function` – ключове слово, яке означає оголошення функції

Для виклику функції достатньо написати її ім'я та `()` у яких можна перерахувати аргументи якщо вони були оголошені

Приклад функції:

```
function f(a1, a2, a3){
    console.log(a1);
    console.log(a2);
    console.log(a3);
}

f(1, 2, 3);
```

Фукнції та процедури

Якщо відходити від теми JS, де синтаксичної різниці між цими поняттями за фактом немає, це дуже важливе питання.

У чому ж різниця?

Функція — блок коду, який в результаті свого виконання поверне якесь закінчення.

Процедура — функція, яка не повертає значення

За повернення значення JS відповідає ключове слово `return`

Давайте розглянемо різницю на прикладах:

```
function func(a1, a2, a3, a4){ // функція
    console.log(a1);
    console.log(a2);
    console.log(a3);
```

```
    return a4;
}

function proc(b1, b2, b3, b4){ // процедура
    console.log(b1);
    console.log(b2);
    console.log(b3);
}

let a = func(1, 2, 3, 4) // == 4(a4)
let b = proc(1, 2, 3, 4) // == undefined
```

Що ж сталося?

При оголошенні **функції** ми використовували `return` тому виклик функції можна буде привласнити змінної і він буде мати значення яке ми укзали в `return`

А при оголошенні **процедури** `return` ми не використовували і через що наш виклик не має значення, тобто `undefined`

Стрілкові функції та привласнення функції змінної

Оскільки функція в JS — це об'єкт, її можна присвоїти змінної.

Тобто оголосити наприклад так:

```
let f = function(a1, a2, a3){
    console.log(a1);
    console.log(a2);
    console.log(a3);
}
```

Такий спосіб не є особливо коректним з логічної точки зору, але при цьому можливим.

Для спрощення запису наведеного вище були придумані так звані стрілочні функції

Синтаксис виглядає приблизно так:

```
let /*ім'я функції*/ = (/*аргумент1*/, /*аргумент2*/.../*аргументN*/) => {
    // блок коду, що виконується
}
```

Є кілька важливих аспектів:

- Якщо в блоці коду лише одна команда `{}` не обов'язкові
- Якщо в блоці коду всього одна команда то вона автоматично повертатиметься через блок `return` навіть не вказуючи його
- Якщо аргумент всього один то `()` не обов'язкові

```
let func = a1 => a1;
let a = func (2); // == 2(a1)
```

Тобто `a1` буде автоматично повертатися і `()` не обов'язкові, оскільки аргумент всього один

Упорядковані типи даних

Змінні - це звичайно добре, але іноді доводиться зберігати багато будь-яких даних об'єднавши їх в якусь одну структуру. Для таких речей вигадані масиви та словники.

Масиви

Масиви — впорядкований набір значень, які можна отримати, звертаючись за індексом.

Масив у JS це об'єкт, так що він має всі властивості об'єкта.

Так само рамотрим кілька способів його оголошення:

```
let arr = новий Array(); // Порожній масив через об'єкт
let arr = new Array(/*елемент 1*/, /*елемент 2*/, ... /*елемент N*/); // Через створення об'єкта, для заповнення
let arr = new Array(/*довжина масиву*/); // Через створення об'єкта, для створення з довжиною за умовчанням
let arr = []; // порожній масив явним чином
let arr = [/*елемент 1*/, /*елемент 2*/, ... /*елемент N*/]; // просто явно
```

Для кращого розуміння скористаємося вище картинкою. Як видно на ній масив - це грубо кажучи набір "осередків" зі значеннями до яких паралельно прикріплені індекси, за якими можна отримати значення, якому вони відповідають.

Для отримання значення або привласнення за індексом достатньо після імені масиву в `[]` вказати індекс.
Для отримання довжини масиву можна викликати властивість `length`.

```
let a = arr [0]; // отримання
arr[0] = 1; // присвоєння
let len = arr.length; // отримання довжини
```

!!!Індесація в мовах програмування починається з 0!!!

Погана та хороша особливість JS:

Якщо в JS звернутися за неоголошеним індексом, то при отриманні значення буде повернено `undefined`, а при привласненні буде створено індекс з цим номером.

Приклад:

```
let arr = [1, 2, 3];
let a = arr [4]; // undefined
arr[4] = 4; // arr == [1, 2, 3, 4]
```

Таке використання не є особливо правильним з логічного погляду, але воно є і насправді з'являється через те, що масив в JS це об'єкт.

Основні методи масивів

- `.concat()` – об'єднання масивів

  ```
  let arr1 = [1, 2, 3];
  let arr2 = [4, 5, 6];
  let arr3 = arr1.concat(arr2); // == [1, 2, 3, 4, 5, 6]
  ```

- `.join(/*подільник*/)` – об'єднує масив у рядок і вставляє між елементами роздільник який ми вкажемо в аргументі

  ```
  let arr = [1, 2, 3];
  let s = arr.join("-"); // == "1-2-3"
  ```

- `.push()` – додавання значення в кінець масиву

  ```
  let arr = [1, 2, 3];
  arr.push(4);
  // arr == [1, 2, 3, 4]
  ```

- `.pop()` – видаляє останній елемент масиву

  ```
  let arr = [1, 2, 3];
  arr.pop();
  // arr == [1, 2]
  ```

- `.unshift()` – додає значення на початок масиву

  ```
  let arr = [1, 2, 3];
  arr.unshift(0);
  // arr == [0, 1, 2, 3]
  ```

- `.shift()` – видаляє перший елемент масиву

  ```
  let arr = [1, 2, 3];
  arr.shift();
  ```

```
// arr == [2, 3]
```

- `.indexOf()` – повертає перший індекс елемента в масиві

```
let arr = [1, 2, 2, 3];
let i = arr.indexOf(2); // 1
```

- `.lastIndexOf()` – повертає останній індекс елемента в масиві

```
let arr = [1, 2, 2, 3];
let i = arr.lastIndexOf(2); // 2
```

- `.slice(/*с*/, /*до*/)` – створення зрізу

```
let arr1 = [1, 2, 3, 4, 5, 6];
let arr2 = arr1.slice (1, 4); // [2, 3, 4]
```

- `.splice(/*індекс*/, /*скільки елементів видалити*/, /*елементи які треба вставити*/)` – видаляє якусь кількість елементів з масиву за індексом і на їх місце вставляє елементи, які перераховані потім

```
let arr = [1, 2, 3, 4, 5, 6];
arr.splice (1, 2, 7, 8);
// arr == [1, 7, 8, 4, 5, 6]
```

- `.reverse()` – переставляє елементи у зворотному порядку

```
let arr = [1, 2, 3];
arr.reverse();
// arr == [3, 2, 1]
```

Про решту можна дізнатися за посиланням:
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array

Перебір масиву

for

```
for(let i = 0; i < arr.length; i ++){
     // елемент масиву arr[i]
}
```

for...in

```
for(let i in arr) {
     // елемент масиву arr[i]
}
```

for...of

```
for(let a of arr){
     // елемент масиву a
}
```

.forEach()

```
arr.forEach(a => {
     // елемент масиву a
});
```

Словники

Іноді виникають ситуації, коли треба, щоб індексами були не цифри, а щось своє. Для цього було створено словники.

Словники – це масив із самозазначеними індексами.

Синтаксис оголошення:

```
let dic = {
     "/*назва індексу 1*/": /*занчення 1*/,
     "/*назва індексу 2*/": /*занчення 2*/,
     ...
     "/*назва індексу N*/": /*занчення N*/,};
```

Звернення та присваєвання таке ж, тільки індекси не обов'язково цифри:

```
let dic = {"name": "Sam", "age": 17}
let d = dic["name"] // "Sam"
dic["age"] = 18;
```

Робота з рядками

Після того, як ми ознайомилися з масивами, можна поговорити про рядки.

По факту: **рядок** – це масив символів. До її елементів можна звертатися як і масиві, але не можна змінювати як елементи в масиві.

```
let s = "abc";
let c = s [0]; // "a"
s[0] = "d"; // неможливо
```

Методи рядків

- `.charCodeAt()` – отримання коду елемента

```
let c = "a".charCodeAt(); // 97
```

- `String.fromCharCode()` – отримання елемента за кодом

```
let c = String.fromCharCode(97); // "a"
```

- `.indexOf()` – повертає перший індекс символу у рядку

```
let s = "abca";
let i = s.indexOf("a"); // 0
```

- `.lastIndexOf()` – повертає останній індекс символу у рядку

```
let s = "abca";
let i = s.lastIndexOf("a"); // 3
```

- `.slice(/*с*/, /*до*/)` – створення зрізу

```
let s1 = "abcdef";
let s2 = s1.slice (1, 4); // "bcd"
```

- `.split(/*подільник*/)` – розбиває рядок на масив рядків по роздільнику

```
let s = "a-b-c";
let arr = s.split("-"); // ["a", "b", "c"]
```

- `.toLowerCase()` – повертає рядок у нижньому регістрі

```
let s1 = "AbC";
let s2 = s1.toLowerCase(); // "abc"
```

- `.toUpperCase()` – повертає рядок у верхньому регістрі

```
let s1 = "AbC";
let s2 = s1.toUpperCase(); // "ABC"
```

- `.repeat()` – повторює рядок кількість разів, що вказано в аргументі

```
let s1 = "abc";
let s2 = s1.repeat(3); // "abcabcabc"
```

- `.trim()` – видаляє прогалини на початку та в кінці рядка

```
let s1 = "abc";
let s2 = s1.trim(); // "abc"
```

Практичні завдання на даних знаннях

4. Придумайте як перевернути рядок у зворотному порядку
5. Знайдіть спосіб рядок `abcdef` перетворити на `cdefgh`
6. Напишіть функцію яка повертатиме рядок створений за правилом `camelCase` (розділювач повинен передаватися другим аргументом)

   Приклад:

   `hello world` -> `helloWorld`
7. Створіть функцію для заповнення масиву словників, які містять інформацію про людину

   Структура:

   ```
   let people = []; // основний масив
   /* структура словника
   {
       "firstName": "",
       "secondName": "",
       "age": "",
       "country": "",
       "city": ""
   };
   */
   ```
8. *Придумайте як можна відсортувати масив

"*" - складна задача

Усе!

Здаэться це все! Я дав вам все, що хотів. Сподіваюся: цей гайд допоміг вам.

Так само у мене є гайд складніший з складнішими темами, але він більш просто збірник рішень, але кому цікаво можете почитати: https://github.com/s0urce18/AdditionalFunctions

Всім дякую! Сподіваюсь ще побачимось :)