**Bootcamp Prep Course**

**Values and Types**

**What is a value?**

* Software stores, retrieves, and manipulates values
* Ex: The number 5 is an example of a value; or ‘apples’ is a value.

// 5 is a value

console.log(‘5 is: ‘, 5);

// apples is a value

console.log(‘apples is: ‘, ‘apples’);

**Values have a type.**

A **type** describes the intrinsic and extrinsic properties of a value that are shared across all values of that type.

For example, the type of 5 is a number. ‘apples’ is a string.

Values of the same type share similar behaviors in JavaScript. e.g., string values can be uppercased, but numbers cannot.

**The typeof operator**

/\* typeof operator returns the type of a value \*/

console.log(typeof 5); // number type

console.log(typeof ‘apples’); // string type

/\* the typeof value returned by typeof - is a string itself \*/

console.log(typeof typeof 5); // this is a string

**Basic arithmetic**

console.log(10 + 5); // returns 15

**Remainder operator**

/\* The % operator divides two numbers and returns the remainder \*/ **modulus**

console.log(10 % 5); // remainder is zero

**Comparing numbers**

/\* use the strictly-equals operator (===) to compare numbers \*/ Returns a boolean

console.log(10 === 10); // returns True

console.log(10 === 11); // Returns False

/\* use the strictly-not-equals operator (!==) \*/

console.log(10 !== 10); // returns False

**Boolean Type**

// only two values have the Boolean type

console.log(typeof true);

console.log(typepof false);

**String type**

// create strings with single quotes or double quotes

console.log(typeof ‘happy’);

**Accessing character**

// strings are a string of characters

// access a character using bracket notation

// the ‘first’ character has an index of 0 (zero)

console.log(‘happy’[0]); // returns h

**The .length property**

console.log(‘happy’.length); // returns 5

**Concatenation**

/\* use the + operator to squish two strings or more together \*/

console.log(‘happy + ‘ ‘ + ‘together’); // returns happy together

**Comparing string values**

// === and !== work with strings too

console.log(‘same’ === ‘same’); // true

console.log(‘same’ !== ‘different’); // true

**Comparing strings with numbers, booleans**

/\* a string will never be strictly equal to a value of a different type \*/

console.log(‘true’ === true); // false

console.log(‘10’ === 10; // false

**String Methods**

* A **method** is an action that can be performed on a value.
* Different types haver different built-in methods

To call a method:

<value>.<name of method>(<args>)

// toUpperCase

console.log(‘so happy’.toUpperCase()); // SO HAPPY

// what if we forget to **invoke** a method? \* notice method doesn’t have () 🡪 not being called

console.log(‘so happy’.toUpperCase); // returns: function toUpperCase()

**practice problems**

1. Write Current Year - Create an Arithmetic Expression that uses three mathematical operators (+,-,/,\*) that returns the date of the current year.

const currentYear = 2000 + (38 / 2) -1;

console.log(currentYear);

**Variables and Expressions:**

**Variable assignment** - values can be stored in a variable by using the assignment operator =

let myNum = 5;

console.log(myNum);

The **let keyword** stores the assigned value in a variable. The variable can be reassigned.

let mood = ‘happy’;

mood = ‘overjoyed’; // reassigned variable ‘mood’ - \*note: doesn’t need **let** keyword

console.log(mood); // returns: overjoyed

The **const** keyword stores the assigned value in a constant variable that CANNOT be reassigned.

const favoriteBootcamp = ‘Fullstack’;

// this line will throw an error

favoriteBootcamp = ‘somewhere else’; // error

**Expressions**

An expression is any valid unit of code that resolves to a value. (MDN) “Mozilla Development Network”.

console.log(15); // returns: 15

console.log(9 === 10); // returns: false

console.log(‘happy’ + ‘ ‘ + ‘together’); // returns: happy together

Since expressions result in a value, expressions can be assigned to variables.

let sum = 10 + 5;

console.log(sum); // returns: 15

**Shortcut syntax**

let sum = 0;

sum = sum + 5;

sum = sum + 10;

console.log(‘sum:’, sum); // returns: sum: 15

The += operator sums and reassigns

let sum = 0;

sum += 5;

sum += 10;

console.log(‘sum’,sum)

The ++ operator increments by 1 and reassigns; and the -- operator decrements by 1.

let myNum = 0;

myNum++;

console.log(‘Fullstack is #’, myNum); // returns: Fullstack is #1

**Test-Driven Development (TDD)**

TDD is a popular approach to writing code among professional engineers:

1. Write tests that run your code and compares the behavior of your code against an expected outcome.
2. Write code that passes the tests.
3. Repeat.

**Some Benefits:**

* Expectations of how code should behave clearly defined before writing code.
* Automated tests easily verify that your code is working as expected.
* Well-written tests help others understand how your code is supposed to work.

**Some perceived drawbacks:**

* Tests take time to write
* Tests can be difficult to read

**TDD plays a central role at Fullstack:**

* Great learning tool
* Very important skillset to develop before entering job market.

**TDD plays a central role in BCP:**

* You will not write tests yourself (which is done for you in boot camp prep)

**Control Flow**

Enables us to control what code to run based on conditions.

**if statement** - starts with the if keyword, then two parentheses around an expression. Finally, curly brackets surrounding the lines of code to run; these lines only run if the expression evaluates to true.

if (true) {

console.log(‘in the if’);

}

**else clause** - the code in an else block will run only if the expression in the **if** evaluates to false.

if (false) {

console.log(‘in the if’);

}

else {

console.log(‘in the else’);

}

**else if clause -** the code in an else if clause has its own condition and only runs if the condition evaluates to true.

if (false) {

console.log(‘in the if’);

}

else if (true) {

console.log(‘in the else if’);

}

example:

let name = ‘Ada’;

if (name === ‘Ada’) {

console.log(‘Hi Countess Lovelace!’);

}

else if (name === ‘Grace’) {

console.log(‘Hi Rear Admiral Hopper!’);

}

else {

console.log(‘Hi friend!’);

}

**Logical Operators**

**&&** is the logical ‘and’ operator. It will return true if placed between two boolean values and both are true.

**||** is the logical ‘or’ operator and returns true if either of the booleans are true.

**Functions**

A **function** is a reusable block of code that works like an equation; you can give them inputs and they can produce outputs, making them an important part of control flow.

syntax

function imAFunction() {

console.log(“Hello world”);

}

// invoking a function

imAFunction();

// parameters and arguments

function hello(name) { // function taking in a parameter

console.log(“Hello ” + name);

}

hello(); // would return: hello undefined

name argument was not passed in, so returns undefined name.

hello(“Gary”); // returns: Hello Gary

// other examples of functions using return

function plus(a, b) {

let sum = a + b;

return sum;

}

function multiply(a,b) {

return a \* b;

}

let x = 3 + plus(10, 5); // 10 + 5 = 15

console.log(x);

let y = multiply(plus(10,5), 2);

^^^ calling the multiply function, passing in the result of the plus function which returns the value of 15, so (15, 2) is passed into the multiply function; returning 15\*2 = 30

////////////

More practicing with return

function isEven(x) {

if (x % 2 === 0) {

return true;

} else {

return false;

}

}

let num = 10;

if (isEven(num)) {

console.log(num + “ is Even”);

} else {

console.log(num + “ is Odd”);

}

**Loops and Debugging**

**While Loop** - requires three elements:

1. the **while** keyword
2. a conditional expression that evaluates to a boolean value
3. a block of code.

while (conditional) {

// block of code

}

let count = 3;

while (count >= 1) {

console.log(‘count is’, count);

count--;

}

while (false) {

console.log(“this will never run”);

}

**For Loop** - requires three elements:

1. the  **for** keyword
2. three optional expressions
3. a block of code

for (initialization; condition; final-expression) {

// block of code

}

the block of code will run over and over until the condition evaluates to false.

The initialization is run first, and only once. It is often used to define a counter variable. Then, before every iteration, the condition is checked to see if it is true.

If it is true, then the for loop will run another iteration. After each iteration, the final expression is run.

for (let i = 1; I <= 3; i++) {

console.log(‘i is: ‘, i);

}

// Use for loops to iterate through a string

let letters = ‘abcdefg’;

for (let i = 0; i < letters.legnth; i++) {

let currentLetter = letters[i];

console.log(currentLetter);

}

The **continue** keyword will cause the loop to skip to the next iteration.

The **break** keyword will exit out of the loop permanently.

**Coercion and Truthiness**

**Coercion** is the way we can explicitly or implicitly change the types of variables.

Explicit is when we define the type of a variable, and implicit is when JS does it for us.

**Explicit coercion**

// number to string

let num = 10;

console.log(typeof num); // returns: number

let numStr = String(num);

console.log(numStr);

console.log(typeof numStr);

// from string to number

let str = “1000”;

console.log(typeof str);

let strNum = Number(str);

console.log(typeof strNum);

console.log(strNum);

**Scope**

Scope refers to which variables can be accessed by your code at a specific location in your code.

**Global scope**

JS is “lexically scoped”, which means the location at which a variable is declared determines its scope.

A variable that is declared outside of a function is globally scoped; it can be referenced from any line of code throughout the file.

Even though a global variable can be referenced from anywhere in your code, the value assigned to the variable cannot be accessed until after the assignment operation occurs.

console.log(waitForIt);

let waitForIt = ‘here I am’;

console.log(waitForIt);

OUTPUT:

undefined

here I am

**Functional Scope**

Variables declared inside of a function are ‘locally-scoped’.

They cannot be referenced outside of the function.

function happyFunction() {

let message = ‘I am so happy!’;

console.log(message);

}

happyFunction();

console.log(message);

OUTPUT:

I am so happy!

ReferenceError: message is not defined

This is because message isn’t defined outside of the function, only inside.

// What if a variable is defined locally and globally?

let message = ‘think globally’;

function logAMessage() {

let message = ‘act locally’;

// JS will look for message locally first.

console.log(message);

}

logAMessage();

if it cant find it locally, JS will look at the scope outside the function, in the above case it is the global scope.

def filter\_list(l):

#'return a new list with the strings filtered out'

my\_list = []

for i in l:

if type(i) == int:

if i >= 0:

my\_list.append(i)

return my\_list

def clock\_angles(hour, minute):

hrHand = 30 \* hour

minuteHand = 6 \* minute

if hrHand < 180:

hrOffset = hrHand

else:

hrOffset = 360 - hrHand

if minuteHand < 180:

minuteOffset = minuteHand

else:

minuteOffset = 360 - minuteHand

difference = hrOffset + minuteOffset

return difference

clock\_angles(3,0)

def sum\_matching\_range(x1, x2, y1, y2):

my\_list1 = []

my\_list2 = []

sum = 0

for i in range(x1, x2+1):

my\_list1.append(i)

#print(my\_list1)

for x in range(y1, y2+1):

my\_list2.append(x)

for num in my\_list1:

if num in my\_list2:

sum += num

return sum

sum\_matching\_range(1, 10, 5, 20)

-----------------------------------------------------------------------------------------------------------------------------------

def liftoff\_countdown(start\_num):

my\_list = []

for x in range(1, start\_num + 1):

my\_list.append(x)

while my\_list:

print(my\_list.pop())

liftoff\_countdown(5)

def fizzbuzz(num):

for x in range(1, num+1):

if x % 3 == 0:

if x % 5 == 0:

print("FizzBuzz")

else:

print("Fizz")

elif x % 5 == 0:

print("Buzz")

else:

print(x)

fizzbuzz(15)