	PYTHON	JAVASCRIPT
1. Compiled vs Interpreted Implementation	Usually **Interpreted**	Usually **Interpreted**
	Interpreted/Compiled is not a property of a language but a property of implementation. In most languages, the implementation falls under one category, however there are exceptions	
	Interpreted Program	
	- code is executed line by line by an interpreter	
	Compiled Program	
	- all code is converted/translated into a lower level machine code before it is run	
2. Statically or Dynamically Typed Language	**Dynamically** Typed	**Dynamically** Typed
	- perform **type checking** at **run-time**	
	- no need to declare variables before you use them	
	Statically Typed	
	- perform type checking at **compile-time**	
	- must declare variables before you use them	
	Type Checking	
	- verifiying if the data types are compatible with the operands being used on them	
	Ex. String + Number ("2" + 3)	
3. Strongly or Weakly Typed Language	**Strongly**-Typed	**Weakly**-Typed
	does NOT allow implicit conversions between unrelated data types	DOES allow implicit conversions between unrelated data types (ex. numbers -> strings)
	Ex.	Ex.
	score = 21	let score = 21;
	score + "3" # TypeError!	score + "3"; //=> "213"
4. Objects	**Everything** is an object	**Almost** everything is an object
		Not Objects: 1) **String**, 2) **Number**, 3) **Boolean**, 4) **Null**, 5) **Undefined**, 6) **Symbol**, 7) **Big Int**
		JS objects are more like Python classes (even though they syntactically look like python dictionaries)
5. Data Types	5 Main Categories:	2 Main Categories:
	1) **Numeric**: **Integer** (ex. 13, -1), **Float** (1.0), **Complex** (ex. 3j)	1) **Primitives**
	2) **Dictionary (ex. { 5: True, "a": 2 })**	2) **Objects**
	3) **Boolean** (ex. True, False)	
	4) **Set** (ex. { "apple", 2, "mango" })	
	5) **Sequence**: **String** ("yes", 'yes'), **List (ex. [1, 2, "a"])**, **Tuple** (ex. (1, "a", ["b", 2]))	
	https://www.geeksforgeeks.org/python-data-types/	
6. Primitive vs Non-Primitive Data Types	No such thing as "primitives" (in the conventional Java / JavaScript sense)	Primitives are the basic building blocks for other data types, and contain a single "value"
		Immutable data types are values that cannot be changed once they are created
		Primitives: 1) **String**, 2) **Number**, 3) **Boolean**, 4) **Null**, 5) **Undefined**, 6) **Symbol**, 7) **Big Int**
		Non Primitives: **Objects**
7. Immutable Data Types	Data type values cannot be changed once they are created	Data type values cannot be changed once they are created
	Immutable Objects:	**Immutable**: 1) **String**, 2) **Number**, 3) **Boolean**, 4) **Null**, 5) **Undefined**, 6) **Symbol**, 7) **Big Int**
	1) **Integer** (ex. 5, -5)	
	2) **Float** (ex. -5.0)	
	3) **Complex** (ex. 3j)	

	PYTHON	JAVASCRIPT
	4) **Tuple** (ex. (1, "a", ["b", 2]))	
	5) **String** (ex. "ye", 'ye')	
	6) **Bytes**	
	7) **Frozen Set**	
	Attempting to change the value of an immutable data type results in error!	Attempting to change the value of an immutable data type does NOT result in error!
	Ex.	Ex.
	name = "max"	let name = "max";
	name[0] = "T" # TypeError! 'str' object does not support item assignment	name[0] = "T"; // no error!
		console.log(name); //=> "max"
8. Variable Declaration/Assignment	No need to declare variable types like in C++ (Ex. int myNum;)	No need to declare variable types like in C++ (Ex. int myNum;)
	No declaration of variable before assignment!	No need to declare variable before assigment, but you can
	Ex.	Ex.
	my_num = 5	let number;
		number = 5;
	No keywords when declaring like in JS (let, const, var)	You can use a keyword before variable (let, const var), to control scope of variable
		let = block scope, reassignable
		const = block scope
		var = function scope, reassignable, redeclarable, hoisted
		no keyword = global scope, reassignable, redeclarable
9. Variable Naming	Same as JS	upper/lowercase letters, numbers, and _
		name cant begin with number
10. Multi Variable Assignment	Ex.	Ex.
	a, b, c = 1, 2, 3	[a, b, c] = [1, 2, 3];
11. Constant Variables	convention is to use all uppercase	use keyword "const"
	constant variables CAN be reassigned	constant variables can NOT be reassigned or redeclared
	Ex.	Ex.
	MY_NUM = 5	const myNum = 5;
	MY_NUM = 10 # ok!	myNum = 10; // TypeError!!!
12. None Data Type	None data type is equivalent to JS "null" data type	
	Ex.	Ex.
	count = None	count = null;
13. Function Hoisting	functions are NOT hoisted	function declarations ARE hoisted
		function expressions are NOT hoisted
14. How to determine a value's data type?	type(___)	typeof ___
	Ex.	Ex.
	type([1, 2, "a"]) # <type 'list'>	typeof [1, 2, "a"] // "object"
15. List Data Type	List data type is equivalent to JS "array"	

	PYTHON	JAVASCRIPT
	Ex.	Ex.
	numbers = [1, 2, 3]	let numbers = [1, 2, 3];
	negative indexing to get items starting from end of list	NO negative indexing supported!
	Ex.	
	numbers = [1, "A", 3]	let numbers = [1, "A", 3];
	print(numbers[-1]) #=> 3	console.log(numbers[-1]); //=> undefined
	indexing items outside the range/doesn't exist results in an ERROR	indexing items outside the range/doesn't exist will NOT result in an error
	Ex.	
	letters = ["a", "b", "c"]	let letters = ["a", "b", "c"];
	letters[3] # ERROR! IndexError: list index out of range	letters[5]; // undefined
16. List Methods/Manipulation- Adding an item	***list*.append(__)**	***array*.push(__, __, ...)**
	- param = any data type (only one param)	- param(s) = item or comma separated items of any data type
	- returns = None	- returns = (num) length of new array
	- adds param item to end of list	- adds param item to end of array
	- will get error you try adding more than one param	
	Ex.	
	nums = [1, 3]	let nums = [1, 3];
	nums.append(5) # returns None	nums.push(5); // returns new array size, 3
	nums # [1, 3, 5]	nums; // [1, 3, 5]
17. List Methods/Manipulation- Combining Lists	**list1 + list2 #=> list3**	**list1 + list2 //=> string**
	- use + plus operator	- concatenates list2 to end of list1 as a string
	- returns new list where items from listB are spread onto the end of listA	- returns string
	Ex.	Ex.
	list1 = [1, 2]	let list1 = [1, 2];
	list2 = ["a", 4]	let list2 = ["a", 4];
	list3 = list1 + list2	let list3 = list1 + list2 // "1,2a,4"
	print(list1, list2, list3) #=> [1, 2] ["a", 4] [1, 2, "a", 4]	
		array1.concat(iterable) //=> new combined array
		use above method for similar python + behavior
18. List Methods/Manipulation- Slicing Lists	**list[start : stop : step] #=> new list**	**array1.slice(startInc, stopExc)**
	- use colon : operator for indexing	- you will get ERROR if you try to use colon : to slice
	- returns new list of sliced items	- returns new array of sliced items
	- start (inclusive) : stop (exclusive) : step (int, optional defaults to 1)	
	Ex.	Ex.
	list1 = ["a", "b", "c", "d"]	let list1 = ["a", "b", "c", "d"]
	list1[0:3] #=> ["a", "b", "c"]	list1.slice(0, 3); //=> ["a", "b", "c"]
	list1[1:-1] #=> ["b", "c"]	list1.slice(0, -1); //=> ["a", "b", "c"]
	list1[1:] #=> ["b", "c", "d"]	list1.slice(0, -2); //=> ["a", "b"]
	list1[:-1] #=> ["a", "b", "c"]	list1.slice(1); //=> ["b", "c", "d"]

	PYTHON	JAVASCRIPT
	list1[:] #=> ["a", "b", "c", "d"]	list1.slice(); //=> ["a", "b", "c", "d"]
		list1.slice(0); //=> ["a", "b", "c", "d"]
	a[start:stop] # items start through stop-1	
	a[start:] # items start through the rest of the array	
	a[:stop] # items from the beginning through stop-1	
	a[:] # a shallow copy of the whole array	
19. Comparison Operators- equality	**same as JS except no strict equality (===), only loose equality (==)**	**loose equality (==) and strict equality (===)**
		- strict equality (===) does NOT perform type coercion if necessary. Ex. 1 === "1" // false
		- loose equality (==) DOES perform type coercion if necessary. Ex 1 == "1" // true
	- equality == operator compares values	- == equality operator for non-primitives (objects) compares memory location NOT values
	Ex1.	Ex1.
	1 == "1" # False	1 == "1" // true
	Ex2.	Ex2.
	[1, 2, "a"] == [1, 2, "a"] # True	[1, 2, "a"] == [1, 2, "a"] // false
20. Arithmetic Operators- Exponent	same as JS **	** or Math.pow(base, exp)
	Ex.	Ex.
	3**2 # 9	3**2 // 9
21. Logical Operators	**and**	**&&**
	or	**||**
	not	**!**
JS BigInt = used to represent large integers (no decimals)		
JS Symbols = used to have private properties in objects, or avoid hash collisions in objects with same keys		
Complex Number = Real Num + Imaginary Num		
Ex. 3 + 2i		
Python Complex Ex.		
z = 3 + 2**j**		
type(z) # <type 'complex'>		
Imaginary Number		
Numbers that when squared are negative		
Needed for modeling electricity, quantum physics, ...		
Irrational Number (ex. pi)		
Python Sets		
- must contain immutable data types		
- duplicate elements not allowed		

	PYTHON	**JAVASCRIPT**
- unordered elements		
Python Tuples		
- immutable (only first level elements)		
- ordered elements		
- can be used as dictionary keys (if tuple only has immutable values)		