Chapter 3: Functions

Defining a function

A function definition is just a regular variable definition where the value given to the variable happens to be a function.

A **function** is created by an expression that starts with the keyword function.

Functions have a set of **parameters** (in this case, only x) and a **body**, which contains the statements that are to be executed when the function is called.

- body must be wrapped in braces, even if only a single statement
- can have multiple parameters or no parameters at all
- some produce a value, some don't

A **return** statement determines the value the function returns. The return keyword without an expression after it will cause the function to return **undefined**.

Parameters and scopes

- parameters are given by **caller** of function, not code in function
- variables created inside, including parameters are **local** to function

Variables declared outside of any function are called **global**

Nested Scope

- each local scope can also see all the local scopes that contain it
- This approach to variable visibility is called **lexical scoping**.

The next version of JavaScript will introduce a **let** keyword, which works like var but creates a variable that is local to the enclosing block, not the enclosing function.

Functions as values

function variables and function values are different.

- function value can be used in an expression, stored in a new place, passed as an argument to a new function, etc.
- function variable can be assigned a new value

Declaration notation

var square = function() is the same as
function square()

This is a function **declaration**.

- function declarations can be defined below code that uses it, because they are conceptually moved to the top of their scope
- don't put function declarations inside (if) block or a loop- only put it in outermost block of a function

The call stack

The place where the computer stores this context is the **call stack**.

- When the stack grows too big, the computer will fail with a message like "out of stack space" or "too much recursion".

Optional Arguments

JavaScript is extremely broad-minded about the number of arguments you pass to a function.

- If you pass too many, the extra ones are ignored.
- If you pass too few, the missing parameters simply get assigned the value undefined.
- this behavior can be used to have a function take "optional" arguments.

Closure

- you can still access and return local variables when the function that created them is no longer active
- multiple instances of the variable can be alive at the same time

Being able to reference a specific instance of local variables in an enclosing function—is called **closure**.

A function that "closes over" some local variables is called a **closure**.

Recursion

A function that calls itself is called **recursive**.

- a simple recursive function is ~10x slower than can equivalent simple loop
- don't worry about efficiency until you know for sure that the program is too slow, at

which point you can start exchanging elegance for efficiency
- some problems *are* easier to solve with recursion, such as those with several branches (i.e. dictionary finding a page)

Growing functions

ways to think about making functions:
- take functionality you've repeated in multiple places, and simplify it by turning it into a function
- you need some functionality and decide it deserves it's own function
  - name the function, then write it's body. maybe even write code using that function before you define the function itself

Functions and side effects

A **pure** function is a specific kind of value-producing function that not only has no side effects but also doesn't rely on side effects from other code