Computability and Algorithms - Course Notes

Formal Languages - Udacity

Course Introduction -(Udacity, Youtube)

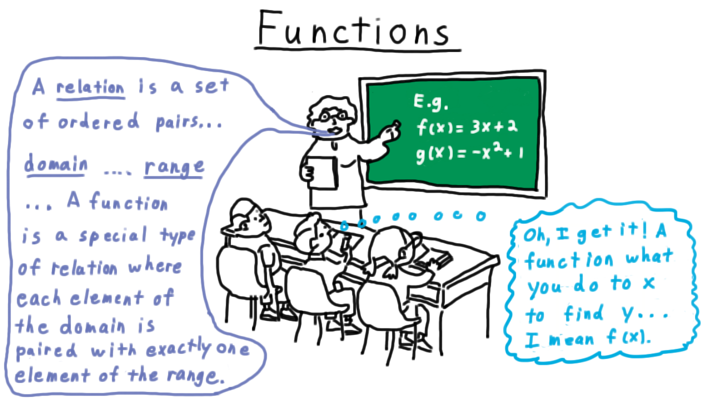
Functions and Computable Functions - (Udacity,Youtube)

In school, it is common to develop a certain mental conception of a function. This conception is often of the form:

> *A function is a procedure you do to an input (often called x) in order to find an output (often called f(x)).*

However, this notion of a function is much different from the mathematical definition of a function:

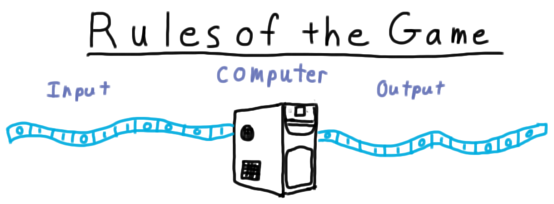
> *A function is a set of ordered pairs such that the first element of each pair is from a set X (called the domain), the second element of each pair is from a set Y (called the codomain or range), and each element of the domain is paired with exactly one element of the range.*


Functions
A relation is a set of ordered pairs... domain .... range ... A function is a special type of relation where each element of the domain is paired with exactly one element of the range.
E.g. f(x)= 3x+2 g(x)= -x²+1
Oh, I get it! A function what you do to x to find y... I mean f(x).

The first conception---the one many of us develop in school---is that of a function as an algorithm: a finite number of steps to produce an output from an input, while the second conception---the mathematical definition---is described in the abstract language of set theory. For many practical purposes, the two definitions coincide. However, they are quite different: not every function can be described as an algorithm, since not every function can be computed in a finite number of steps on every input. In other words, there are functions that computers cannot compute.

Rules of the Game - (Udacity, Youtube)

When you hear the word computation, the image that should come to mind is a machine taking some input, performing a sequences of operations, and after some time (hopefully) giving some output.

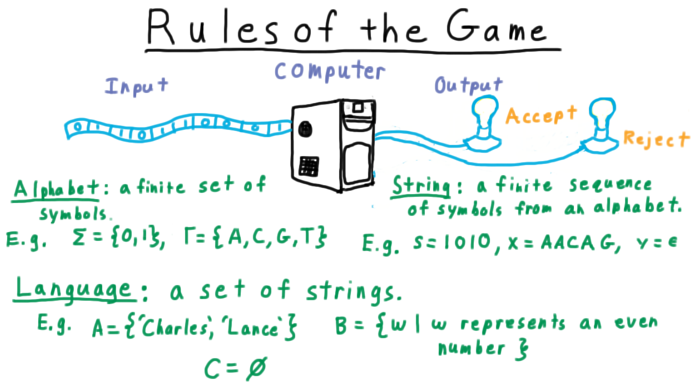


In this lesson, we will focus on the forms of the inputs and output and leave the definition of the machine for later.

The inputs read by the machine must be in the form of *strings* of characters from some finite set, called the machine's *alphabet*. For example, the machine's alphabet might be binary (0s and 1s), it might be based on the genetic code (with symbols A, C, G, T), or it might be the ASCII character set.

Any finite sequence of symbols is called a *string*. Thus, '0110' is a string over the binary alphabet. Strings are the only input data type that we allow in our model.

Sometimes, we will talk about machines having string outputs just like the inputs, but more often than not, the output will just be binary--an up or down decision about some property of the input. You might imagine the machine just turning on one of two lights, one for accept, or one for reject, once the machine is finished computing.


Rules of the Game
Input
Computer
Output
Accept
Reject
Alphabet: a finite set of symbols.
E.g. Σ = {0,1}, Γ = {A,C,G,T}
String: a finite sequence of symbols from an alphabet.
E.g. S = 1010, X = AACAG, Y = ε
Language: a set of strings.
E.g. A = {'Charles', 'Lance'} B = {w | w represents an even number}
C = ∅

With these rules, an important type becomes a collection of strings. Maybe, it's the set of strings that some particular machine accepts, or maybe we are trying to design a machine so that it accepts strings in a certain set and no others, or maybe we're asking if it's even possible to design a machine that accepts everything in some particular set and no others. In all these cases, it's a set of strings that we are talking about, so it makes sense to give this type its own name.

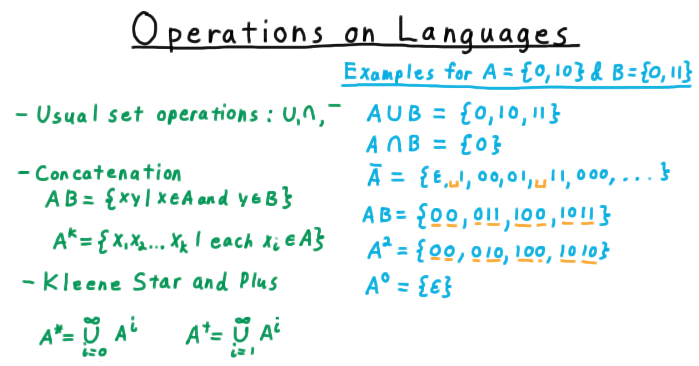
We call a set of strings a *language*. For example, a language could be a list of names, It could be the set of binary strings that represent even numbers--notice that this set is infinite--or it could be the empty set. *Any* set of strings over an alphabet is a language.

Operations on Languages - (Udacity, Youtube)

The concept of a language is fundamental to this course, so we'll take a moment to describe common operations used to manipulate languages. For examples we'll use the languages $$$A = $$${'0',10'} and $$$B = $$$ {'0','11'} over the zero-one alphabet.

Since languages are sets, we have the usual operations of union, intersection and complement defined for them. For example $$$A$$$ union $$$B$$$ consists of the three strings '0','10', and '11'. The string '0' comes from both $$$A$$$ and $$$B$$$, '10' from $$$A$$$ and '11' from $$$B$$$. The intersection contains only those strings in both languages, just the string '0'.

To define the complement of a language we need to make clear what it is that we are completing. It's not sufficient just to say that it is everything not in $$$A$$$. For $$$A$$$ *that* would include strings with characters besides 0 and 1 or maybe even infinite sequences of 0's and 1's, which we don't want. The complement, therefore, is defined so as to complete the set of all strings over the relevant alphabet, in this case the binary alphabet. The alphabet over which the language is defined is almost always clear from context. In this case, the complement of $$$A$$$ will be infinite.

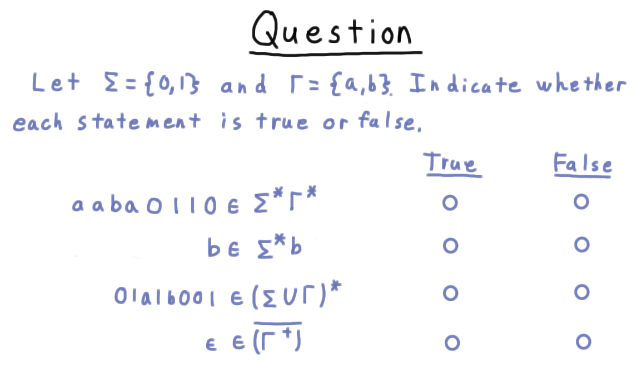

Operations on Languages
Examples for $A = \{0,10\}$ & $B = \{0,11\}$
- Usual set operations : $\cup, \cap, \overline{}$
$A \cup B = \{0,10,11\}$
$A \cap B = \{0\}$
$\bar{A} = \{\varepsilon, 1, 00, 01, 11, 000, \ldots\}$
- Concatenation
$AB = \{xy \mid x \in A \text{ and } y \in B\}$
$A^k = \{x_1 x_2 \ldots x_k \mid \text{each } x_i \in A\}$
$AB = \{00, 011, 100, 1011\}$
$A^2 = \{00, 010, 100, 1010\}$
$A^0 = \{\varepsilon\}$
- Kleene Star and Plus
$A^* = \bigcup_{i=0}^{\infty} A^i$
$A^+ = \bigcup_{i=1}^{\infty} A^i$

In addition to these standard set operations, we also define an operation for concatenating two languages. The concatenation of $$$A$$$ and $$$B$$$ is just all strings you can form by taking a string from $$$A$$$ and appending a string from $$$B$$$ to it. In our examples, this set would be '00' with first 0 coming from $$$A$$$ and second from $$$B$$$. The string '011' with the '0' coming from $$$A$$$ and the '11' coming from $$$B$$$, and so forth. Of course, we can also concatentate a language with itself. Instead of writing $$$AA$$$, we often write $$$A^2$$$. In general, when we want to concatenate a language with itself k times we write $$$A$$$ to the kth power. Note that for $$$k=0$$$, this defined as the language containing exactly the empty string.

When we want to concatenate *any* number of strings from a language together to form a new language, we use an operator known as Kleene star. This can be thought of as the union of all possible powers of the language. When we want to exclude the emptry string, we use the plus operator, which insists that at least one string from A be used. Notice the difference in starting indices. So for example, the string '01001010' is in $$$ A^*$$$. There is a way that I can break it up so that each part is in the language $$$A$$$, and so as a whole the string can be thought of a concatenation of strings from $$$A$$$. Note that even $$$A^*$$$ doesn't include infinite sequences of symbols. Each individual string from A must be of finite length, and you are only allowed to concatenate a finite number together.

For those who have studied regular expressions, this should seem quite familiar. In fact, one gets the notation of regular expressions by treating individual symbols a languages. For example, $$$0^*$$$ is the set of all strings consisting entirely of zeros. Here we are treating the symbol 0 as a language unto itself. We will also commonly refer to $$$\Sigma^*$$$, meaning all possible strings over the alphabet $$$\Sigma$$$. Here we are treating the individual symbols of the alphabet as strings in the language $$$\Sigma$$$.

Language Operations Exercise - (Udacity)



Question

Let $\Sigma = \{0,1\}$ and $\Gamma = \{a,b\}$. Indicate whether each statement is true or false.

	True	False
$aaba0110 \in \Sigma^*\Gamma^*$	○	○
$b \in \Sigma^* b$	○	○
$01a1b001 \in (\Sigma \cup \Gamma)^*$	○	○
$\epsilon \in \overline{(\Gamma^+)}$	○	○

Countability (Part 1) - (Udacity, Youtube)

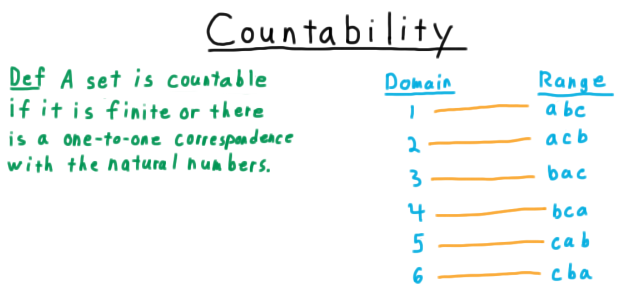
We need one more piece of mathematical background before we can more formally prove our claim that not all functions are computable. Intuitively, the proof will show that there are more languages than there are possible machines. The set of possible computer programs is *countably* infinite, but the set of languages over an alphabet in *uncountably* infinite.

If you aren't familiar with the distinction between countable and uncountable sets already, you may be thinking to yourself "inifinity is a strange enough idea by itself; now I have to deal with two of them!" Well, it isn't as bad as all that. A countably infinite set is one that you can enumerate--that is, you can say this is the first element, this is the second element, and so forth, *and* --this is the really important part-- eventually give a number to every element of the set. For some sets an enumeration straightforward. Take the even numbers. We can say that 2 is the first one, 4 the second, 6 the third and so forth. For some sets, like the rationals you have to be a little clever to find an enumeration. We'll see the trick for enumerating them in a little bit. And for some sets, like the real numbers, it doesn't matter how clever you are; there simply is no enumeration. These are the uncountable sets.

Let us make the following definition:

> *A set S is countable if it is finite or if it is in one-to-one correspondence with the natural numbers.*

A one-to-one correspondence, by the way is a function that is one-to-one, meaning that no two elements of the domain get mapped to the same element of the range, and also onto, meaning the every element of the range is mapped to by an element of the domain. For example, here is a one-to-one correspondence between the integers 1 through 6 and the set of permutations of three elements.



Countability

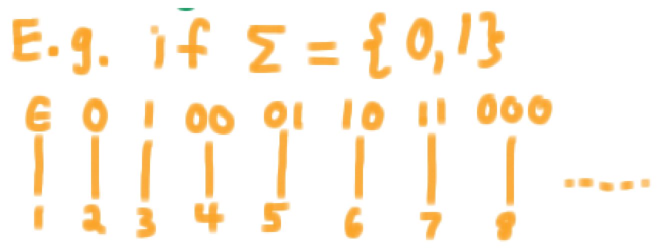
Def A set is countable if it is finite or there is a one-to-one correspondence with the natural numbers.

Domain	Range
1	abc
2	acb
3	bac
4	bca
5	cab
6	cba

Now, in general, the existence of a one-to-one correspondence implies that the two sets have the same size-- that is, the same number of elements. And this actually holds even for infinite sets. This is why we say that there are as many even natural numbers as there are natural numbers: because it's easy to establish a one-to-one correspondense between the two sets, f(n) = 2n for example. Some examples of *countably infinite* sets are:

- The set of nonnegative even numbers (a one-to-one correspondence to the natural numbers is $$$f(x) = x/2)$$$
- The set of positive odd numbers (with correspondence $$$f(x) = (x-1)/2$$$
- The set of all integers (with correspondence $$$f(x) = 2x$$$ if $$$x$$$ is nonnegative and $$$f(x) = -2x - 1$$$ if $$$x$$$ is negative)

For our purposes, we want to show that the set of all strings over a finite alphabet is countable. Since computer programs are always represented as finite strings, this will tell us that the set of computer programs is countable. The proof is relatively straightforward. Recall that $$$\Sigma*$$$ is the union of strings of size 0 with those of size 1 with those of size 2, etc. Our strategy will just be to assign the first number to $$$\Sigma^0$$$, the next numbers $$$\Sigma^1$$$, the next to $$$\Sigma^2$$$, etc. Here is the enumeration for the binary alphabet.



The key is that every string gets a positive integer mapped to it under this scheme. Therefore,

> *The set of all strings over any alphabet is countable.*

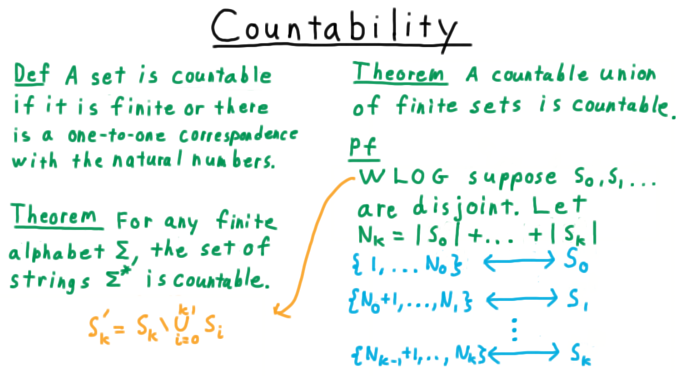
Countability (Part 2) - (Udacity, Youtube)

This same argument shows that

> *A countable union of finite sets is countable.*

Suppose that our set of sets is $$$S_0, S_1,...$$$ etc. Without loss of generality, we'll suppose that they are disjoint. (If they happen not to be disjoint, we can always make them so by subtracting out from $$$S_k$$$ all the elements it shares with sets $$$S_0...S_k-1$$$.)

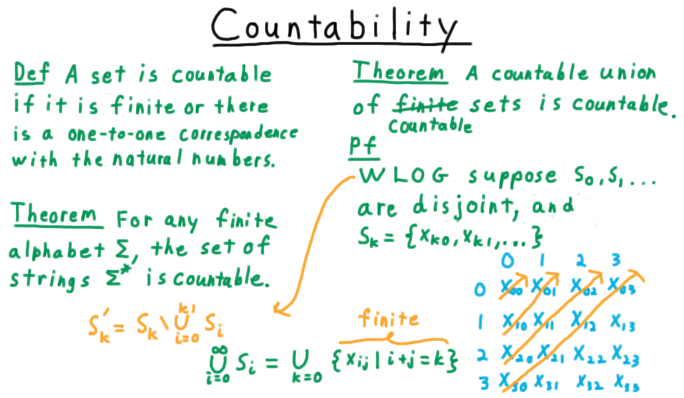
Then the argument proceeds just as before. We assign the first numbers to $$$S_0$$$, the next to $$$S_1$$$, etc. Every element in the union must have a first set $$$S_k$$$ that it belongs to, and thus it will be counted in the enumeration.


Countability
Def A set is countable if it is finite or there is a one-to-one correspondence with the natural numbers.
Theorem For any finite alphabet Σ, the set of strings Σ* is countable.
$S_k' = S_k \setminus \bigcup_{i=0}^{k-1} S_i$
Theorem A countable union of finite sets is countable.
Pf WLOG suppose $S_0, S_1, \ldots$ are disjoint. Let $N_k = |S_0| + \ldots + |S_k|$
$\{1, \ldots N_0\} \longleftrightarrow S_0$
$\{N_0+1, \ldots, N_1\} \longleftrightarrow S_1$
$\{N_{k-1}+1, \ldots, N_k\} \longleftrightarrow S_k$

It turns out that we can actually prove something even stronger than this statement here. We can replace this word finite with the word countable, and say that

> A countable union of countable sets is countable.

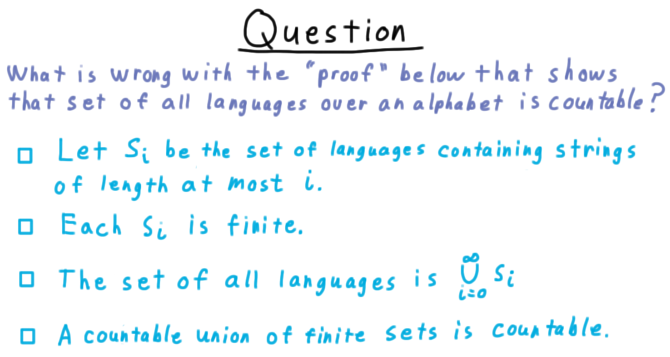
Notice that our current proof doesn't work. If we tried to count all of the elements of $$$S_0$$$ before any of the elements of $$$S_1$$$, we might never get to the elements of $$$S_1$$$, or any other set besides $$$S_0$$$. Nevertheless, this theorem is true. For convenience of notation, we let the elements of $$$S_k$$$ be $$${x_{k0}, x_{k1},…}$$$ and then we can make each set $$$S_k$$$ a row in a grid.


Countability
Def A set is countable if it is finite or there is a one-to-one correspondence with the natural numbers.
Theorem For any finite alphabet Σ, the set of strings Σ* is countable.
$S_k' = S_k \setminus \bigcup_{i=0}^{k-1} S_i$
Theorem A countable union of ~~finite~~ countable sets is countable.
Pf WLOG suppose $S_0, S_1, \ldots$ are disjoint, and $S_k = \{x_{k0}, x_{k1}, \ldots\}$
finite
$\bigcup_{i=0}^{\infty} S_i = \bigcup_{k=0} \{x_{ij} \mid i+j=k\}$
x_{00} x_{01} x_{02} x_{03}
x_{10} x_{11} x_{12} x_{13}
x_{20} x_{21} x_{22} x_{23}
x_{30} x_{31} x_{32} x_{33}

Again, we can't enumerate row-by-row here because we would never finish the first row. On the other hand, can go diagonal-by-diagonal, since each diagonal is finite. The union of all the set S_k is the union of all the rows, but that is the same as the union of all the diagonals. Each diagonal being finite, we can then apply the original version of the theorem to prove that a countable union of countable sets is countable.

Note that his idea proves that the rationals are countable. Imagine putting all fractions with a 1 in the numerator in the first row, all those with a 2 in the numerator in the second row, etc.

A False Proof - (Udacity)



Question

What is wrong with the "proof" below that shows that set of all languages over an alphabet is countable?

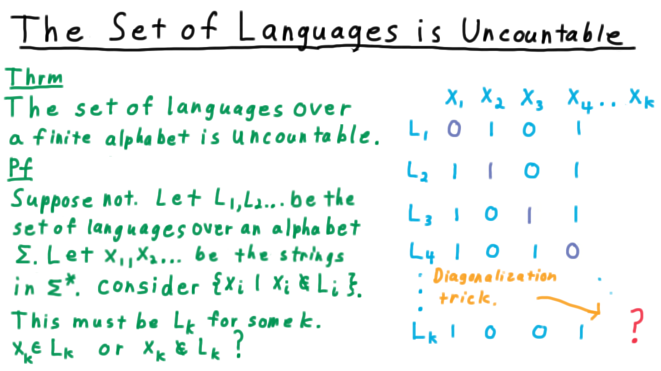
- ☐ Let S_i be the set of languages containing strings of length at most i.
- ☐ Each S_i is finite.
- ☐ The set of all languages is $\bigcup_{i=0}^{\infty} S_i$
- ☐ A countable union of finite sets is countable.

Languages are Uncountable - (Udacity, Youtube)

So far, we've seen that the set of strings over an alphabet in countable. But what about all subsets of these strings? What about the set of all languages. It turns out that this set is *uncountable*.

> *The set of all languages over an alphabet is uncountable.*

For the proof, we'll suppose not. That is, suppose there is an enumeration of the languages $$$L_1, L_2, \ldots $$$ over an alphabet $$$\Sigma$$$. Also, let $$$x_1, x_2,\ldots $$$ be the strings in $$$\Sigma*$$$. We are then going to build a table, where the columns correspond to the strings from Sigma* and the rows correspond to the languages. In each entry in the table, we'll put a 1 if the string is in the language and a 0 if it is not.



The Set of Languages is Uncountable

Thrm
The set of languages over a finite alphabet is uncountable.

Pf
Suppose not. Let $L_1, L_2 \ldots$ be the set of languages over an alphabet Σ. Let $x_1, x_2 \ldots$ be the strings in Σ^*. Consider $\{x_i \mid x_i \notin L_i\}$. This must be L_k for some k. $x_k \in L_k$ or $x_k \notin L_k$?

	x_1	x_2	x_3	x_4 ..	x_k
L_1	0	1	0	1	
L_2	1	1	0	1	
L_3	1	0	1	1	
L_4	1	0	1	0	
⋮					
L_k	1	0	0	1	?

Now, we are going to consider a very sneaky language defined as follows: it consists of the those strings $$$x_i$$$ for which $$$x_i$$$ is not in the language $$$L_i$$$. In effect, we've taken the diagonal in this table and just reversed it. Since we are assuming that the set of languages is countable, this language must be $$$L_k$$$ for some $$$k$$$. But is $$$x_k$$$ in $$$L_k$$$ or not? From the table, the row $$$L_k such$$$ that in every column the entry is the opposite of what is on the diagonal. But the diagonal entry can't be the opposite of itself.

If $$$x_k$$$ is in $$$L_k$$$, then according to the definition of $$$L_k$$$, it should not be in $$$L_k$$$. On the other hand, if $$$x_k$$$ is not in $$$L_k$$$, then it should be in $$$L_k$$$. Another way to think about this argument is to say that this oppositie-of-the-diagonal language must be different from every row in the table because it is different from the diagonal element. In any case, we have a contradiction and can conclude that this enumeration of the languages was invalid since the rest of our reasoning was sound. This argument here is known as the diagonalization trick, and we'll see it come up again later, when we discuss Undecidability.

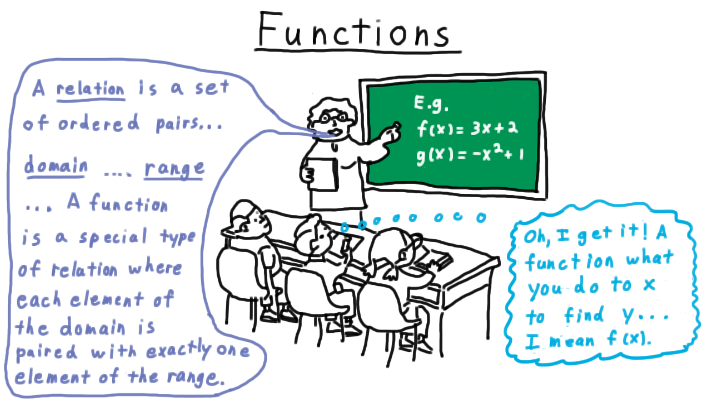
Consequences - (Udacity, Youtube)

Although they may not be immediately apparent, the consequences of the uncountability of languages are rather profound. We'll let $$$\Sigma$$$ be the set of ASCII characters--these are all the ones you would need to program-- and observe that the set of strings that represent valid python programs is a subset of $$$\Sigma^*$$$. Each program is a finite string, after all. (The choice of python arbitrarily-- any language or set of languages works.) Since this set is subset of the countable set $$$\Sigma^*$$$, we have it is countable. Thus, there are a countable number of python programs.

On the other hand, consider this fact. For any language $$$L$$$, we can define $$$F_L$$$ to be the function that is 1 if x is in $$$L$$$ and 0 otherwise. All such functions are distinct, so the set of these functions must be uncountable, just like the set of all languages.

Here is the profound point, since the set of valid python programs is countable but the set of functions is not, it follows that there must be some functions that we just can't write programs for. In fact, there are uncountably many of them!

So going back to our picture of the HS classroom, we can see that the teacher, perhaps without realizing it was talking about something *much* more general that what the student ended up thinking. There are only countably many computer programs that can follow a finite number of steps as the student was thinking, but there are uncountably many functions that fit the teachers definition.


Functions
A relation is a set of ordered pairs... domain .... range ... A function is a special type of relation where each element of the domain is paired with exactly one element of the range.
E.g. f(x) = 3x+2 g(x) = -x²+1
Oh, I get it! A function what you do to x to find y... I mean f(x).