Chapter 6

A Taste of Computability Theory

6.1 The problem

Algorithms (implemented as computer programs) can carry out many complex tasks. Some of the more interesting ones concern computers and programs themselves, *e.g.*, compiling, interpreting, and other manipulations of source code.
Here is one particular task that we would like to carry out.

```
def halt(f,i):
    """Return True iff f(i) eventually halts."""

    return True  # replace this stub with correct code
```

Note that function `halt` is well-defined: `halt` is passed a reference to some other Python function `f` along with an input `i`, and it must return `True` if `f(i)` eventually halts (normally, or because of a crash); `False` if `f(i)` eventually gets stuck in some infinite loop. These are the only two possible behaviours for the call `f(i)` — we ignore any possibility of hardware failure (which could not be detected from within `halt` anyway). In other words, we are interested in the *conceptual* behaviour of `f` itself, under ideal conditions for its execution.

Before you read the next section, see if you can come up with an implementation that works — even if it's just at a high-level and not fully written out in Python. As a guide, think about the value returned by your code for the call `halt(blah,5)`, or for the call `halt(blah,8)`, where `blah` is the following function.

```
def blah(x):
    if x % 2 == 0:
        while True:  pass
    else
        return x
```

6.2 An impossible proof

What if we told you that it is impossible to implement function `halt`? Note that this is a very strong claim to make: we're NOT just saying "we don't know how to write `halt`"; we're saying "NOBODY can write `halt`, ever, because it simply cannot be done"!

Because this is such a strong claim, it is daunting to prove. Also, how can we even prove that something is *not possible*? Wouldn't that require us to argue about every possible way to try to carry out this task?[1]

Yet, we will be able to do just that using only the proof techniques we've seen so far. Here is how...

Assume `halt` exists (*i.e.*, it is fully written out in Python).
Then consider the following function.

```
def confused(f):
    def halt(f,i):
        ...copy/paste the full code for halt here...

    if halt(f,f):                # line 1
        while True:  pass        # line 2
    else:
        return False             # line 3
```

Note that this is a correct Python function (assuming we've pasted in the code for `halt`).[2]
Now, we ask: what is the behaviour of the call `confused(confused)`?
Either `confused(confused)` halts or `confused(confused)` does not halt.[3]
Case 1: Assume `confused(confused)` halts.
Then `halt(confused,confused)` returns `True` (on line 1). # by definition of `halt`
Then `confused(confused)` goes into an infinite loop (on line 2).
So `confused(confused)` halts $\Rightarrow$ `confused(confused)` does not halt.
Case 2: Assume `confused(confused)` does not halt.
Then `halt(confused,confused)` returns `False` (on line 1). # by definition of `halt`
Then `confused(confused)` returns `False` (on line 3).
So `confused(confused)` does not halt $\Rightarrow$ `confused(confused)` halts.
Hence, `confused(confused)` halts $\Leftrightarrow$ `confused(confused)` does not halt. # $\Leftrightarrow$I
Clearly, this is a contradiction. # of the form $p \Leftrightarrow \neg p$
Then, by contradiction, `halt` does not exist!

This is such a counter-intuitive result that it is tempting to dismiss it at first. But take the time to think through each step of the proof, and you will see that they are all correct. Moreover, this result does NOT expose a weakness in Python: the exact same argument would apply to every possible programming language (in fact, to things that we would not even call "programming languages," as we discuss next).

HISTORICAL CONTEXT

The proof we just showed was discovered by the mathematicians Alan Turing and Alonzo Church (independently of each other), back in the late 1930's — before computers even existed! They argued that every possible algorithm can be expressed using a small set of primitive operations ("λ-calculus" in the case of Church and "Turing Machines" in the case of Turing). Then, they carried out the argument above based on those primitive operations.

If you believe that Python is capable of expressing every possible algorithm (and it is), then the argument shows that some problems cannot be solved by any algorithm. (Think about it: what features of Python did we need in our proof?[4])

6.3 COUNTABILITY

The idea for function `confused` in the proof above is an example of a DIAGONALIZATION argument, first used by the mathematician Georg Cantor to show that the set of real numbers is larger than the set of natural numbers.

Now that, in itself, may seem like a strange statement to make: how can the set of real numbers be "larger" than the set of natural numbers? Don't they both contain infinitely many numbers? What does it even mean to compare the sizes of infinitely large sets?

Let's take things one step at a time and scale back to simpler infinite sets. Consider $\mathbb{N}$ (the set of natural numbers) and $\mathbb{Z}$ (the set of integers). Between the two, which set is LARGER? The answer seems obvious: $\mathbb{Z}$ contains $\mathbb{N}$ as proper subset, so $\mathbb{Z}$ must be larger.

But now think about this: which set has larger SIZE? Now the answer is not so obvious: for finite sets "size" is a natural number, but how do we measure the size of infinite sets? Adding or removing one element from an infinite set does not change its size: it's still infinite. So are all infinite sets the same size? And what does "infinite size" even mean?

Let's think through this more carefully. When we COUNT the elements in a set, what we are really doing is *associating a number with each element*: you can easily imagine yourself pointing to elements one after the other while saying "one, two, three, ..." The easiest way to formalize this idea involves the notions of one-to-one and onto functions.

DEFINITION: Suppose $f : A \to B$ (*i.e.*, f is a function that associates an element $f(a) \in B$ to each element $a \in A$). Then we say that

- f is ONE-TO-ONE if $\forall a_1 \in A, \forall a_2 \in A, f(a_1) = f(a_2) \Rightarrow a_1 = a_2$ (*i.e.*, f gives distinct values to different elements of A);
- f is ONTO if $\forall b \in B, \exists a \in A, f(a) = b$ (*i.e.*, every element in B can be "reached" from at least one element of A).

For example,

- the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(x) = \lfloor x/2 \rfloor + 2$ is neither one-to-one ($f(2) = 3 = f(3)$) nor onto (there is no $x \in \mathbb{N}$ such that $f(x) = 1$);
- the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(x) = 2x$ is one-to-one ($f(x) = f(y) \Rightarrow x = y$) but not onto (there is no $x \in \mathbb{N}$ such that $f(x) = 1$);
- the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(x) = \lfloor x/2 \rfloor$ is not one-to-one ($f(2) = 1 = f(3)$) but it is onto (for all $n \in \mathbb{N}$, $f(2n) = n$);
- the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(x) = 10 - x$ if $x \leqslant 10$; $f(x) = x$ otherwise is both one-to-one (the numbers $\{0, \ldots, 10\}$ get mapped to $\{10, \ldots, 0\}$ and $\{11, 12, \ldots\}$ remain unchanged) and onto ($\forall n \in \mathbb{N}, (n \leqslant 10 \Rightarrow f(10 - n) = n) \wedge (n > 10 \Rightarrow f(n) = n)$).

(You can find out more about one-to-one and onto functions in Section 5.2 of Velleman's "How to Prove It," or on Wikipedia.)
Using these concepts, the idea of "counting" can be formalized as follows:

DEFINITION: Set A is COUNTABLE if

1. there is a function $f : A \to \mathbb{N}$ that is one-to-one, or equivalently,
2. there is a function $f : \mathbb{N} \to A$ that is onto.

For example,

CLAIM 6.1: $\mathbb{Z}$ is countable.

Let

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even,} \\ (1-n)/2 & \text{if } n \text{ is odd.} \end{cases}$$

Then $f : \mathbb{N} \to \mathbb{Z}$.
Next, we show that f is onto. Assume $x \in \mathbb{Z}$.

Then $x \leqslant 0$ or $x \geqslant 0$.
Case 1: Assume $x \leqslant 0$.
Let $n' = 1 - 2x$.
Then $x \leqslant 0 \Rightarrow -2x \geqslant 0 \Rightarrow n' \geqslant 0$ so $n' \in \mathbb{N}$.
Also, $n' = 2(-x) + 1$ so n' is odd.
Then $f(n') = (1 - n')/2 = (1 - (1 - 2x))/2 = 2x/2 = x$.
Hence $\exists n \in \mathbb{N}, f(n) = x$.
Case 2: Assume $x \geqslant 0$.
Let $n' = 2x$.
Then $x \geqslant 0 \Rightarrow n' \geqslant 0$ so $n' \in \mathbb{N}$.
Also, $n' = 2x$ is even.
Then $f(n') = n'/2 = 2x/2 = x$.
Hence $\exists n \in \mathbb{N}, f(n) = x$.
In all cases, $\exists n \in \mathbb{N}, f(n) = x$.
Since x was arbitrary, $\forall x \in \mathbb{Z}, \exists n \in \mathbb{N}, f(n) = x$, *i.e.*, f is onto. (Note that f is not one-to-one — can you see why?[5])
Hence, there is a function $f : \mathbb{N} \to \mathbb{Z}$ that is onto, *i.e.*, $\mathbb{Z}$ is countable.

Informally, we often show that a set A is countable by giving an argument that it is possible to LIST every element in the set — this corresponds to giving a function $f : \mathbb{N} \to A$ that is onto, though the function may not be written out algebraically. What matters most is to make sure that there is some systematic way to list the elements of A and that the list includes every element of A at least once.

For example, we may argue that $\mathbb{Z}$ is countable by exhibiting the following list (really, more a *list pattern* because of the "..."):

$$\mathbb{Z} : 0, -1, 1, -2, 2, -3, 3, \ldots$$

Implicitly, this defines a function $f : \mathbb{N} \to \mathbb{Z}$ — just think of the list as an enumeration of f's values ($f(0) = 0, f(1) = -1, f(2) = 1, \ldots$) Once we see the pattern, it is obvious that the list includes every possible integer, *i.e.*, the implicit function f is onto. Hence, $\mathbb{Z}$ is countable.

What about finite sets, *e.g.*, $\{a, b, c\}$? Is it countable? Yes, because of condition 1 in the definition of countability: the function $f(a) = 1$, $f(b) = 2$, $f(c) = 3$ is easily proved to be one-to-one.

Assume $x, y \in \{a, b, c\}$ and $f(x) = f(y)$.
Then $f(x) = f(y) = 1$ or $f(x) = f(y) = 2$ or $f(x) = f(y) = 3$.
\# these are the only values in the domain of f
Case 1: Assume $f(x) = f(y) = 1$.
Then $x = y = a$ so $x = y$.
Case 2: Assume $f(x) = f(y) = 2$.
Then $x = y = b$ so $x = y$.
Case 3: Assume $f(x) = f(y) = 3$.
Then $x = y = c$ so $x = y$.
In every case, $x = y$.
Hence $\forall x \in \{a, b, c\}, \forall y \in \{a, b, c\}, f(x) = f(y) \Rightarrow x = y$, *i.e.*, f is one-to-one.

What about other infinite sets? Are they all countable? Consider the set $\mathbb{Q}$ of rational numbers (*i.e.*, numbers of the form p/q for integers p, q with $q \neq 0$). Numerically, we know that this set is much different from $\mathbb{Z}$ or $\mathbb{N}$: the rational numbers are DENSE (there are infinitely many rational numbers between any two other rational numbers). This difference might seem to imply that $\mathbb{Q}$ is not countable, for how could we hope to list all of the rational numbers? However, note that our informal definition of countability does NOT require that we be able to list the elements in any kind of numerical order — in fact, our list for $\mathbb{Z}$ above was not ordered numerically.

In order to show that $\mathbb{Q}$ is countable, we first prove a lemma.

Claim 6.2: $\mathbb{Q}^+$ is countable (where $\mathbb{Q}^+$ is the set of *positive* rational numbers).

We give an informal "list" argument.
Let $f(n)$ be defined implicitly by the following list process: list fractions p/q in increasing order of $p+q$, starting with $p+q=2$, and in increasing order of p within each sub-list with a fixed value of $p+q$. The first few terms of the list are as follows:

- sub-list 0: $1/1$, (fractions p/q where $p+q=2$)
- sub-list 1: $1/2, 2/1$, (fractions p/q where $p+q=3$)
- sub-list 2: $1/3, 2/2, 3/1$, (fractions p/q where $p+q=3$)
- ...

Then $f(n) : \mathbb{N} \to \mathbb{Q}^+$. Also, $f(n)$ is onto: every fraction p/q with $p>0, q>0$ is eventually listed (in position p of sub-list number $p+q-2$). Hence, by definition, $\mathbb{Q}^+$ is countable.

Claim 6.3: $\mathbb{Q}$ is countable.

Note that $\mathbb{Q} = \{0\} \cup \mathbb{Q}^- \cup \mathbb{Q}^+$.
Let $f(n)$ be the function from lemma 6.2 (showing $\mathbb{Q}^+$ is countable).
Then, the following is a complete list of $\mathbb{Q}$, *i.e.*, it defines a function $f' : \mathbb{N} \to \mathbb{Q}$ that is onto:

$$\begin{aligned} &0, f(0), -f(0), f(1), -f(1), f(2), -f(2), \ldots \\ &= 0, 1/1, -1/1, 1/2, -1/2, 2/1, -2/1, \ldots \end{aligned}$$

By this point, you may start to think that EVERY set is countable, and that the notion is meaningless. However, this turns out not to be the case. To see this, first try to come up with a complete list of $\mathbb{R}$ (the set of real numbers) — to make it easier, you may begin with just $\mathbb{R}^+$, as we did for $\mathbb{Q}$.
If you run into difficulties, remember that

- every real number $r \in \mathbb{R}$ can be written as an infinite decimal expansion of the form $r = m.d_1d_2d_3\ldots$, where $m \in \mathbb{Z}$ and $d_1, d_2, d_3, \ldots \in \{0,1,2,3,4,5,6,7,8,9\}$ and the expansion does NOT end with repeating 9's;
- every infinite decimal expansion of the form $r = m.d_1d_2d_3\ldots$, where $m \in \mathbb{Z}$ and $d_1, d_2, d_3, \ldots \in \{0,1,2,3,4,5,6,7,8,9\}$ and the expansion does NOT end with repeating 9's, defines a unique real number $r \in \mathbb{R}$.

The provision that the decimal expansion does not end with repeating 9's is a consequence of the positional notation system, and the fact that $0.\overline{9}\ldots = 1.\overline{0}\ldots$ Hence, every decimal expansion that ends with infinitely many 9's is equivalent to a decimal expansion that ends with infinitely many 0's instead — but all other decimal expansions are different from each other.

6.4 Diagonalization

Cantor's proof

Try as you might, you will not be able to provide a complete list of $\mathbb{R}$. Georg Cantor showed that this was impossible through the following proof, called a DIAGONALIZATION ARGUMENT.

Claim 6.4: $\mathbb{R}$ is uncountable.

PROOF:

For a contradiction, assume that $\mathbb{R}$ is countable.

Then there is a function $f : \mathbb{N} \to \mathbb{R}$ that is onto. # by definition of "countable"

We can visualize function f as follows: for every natural number $n \in \mathbb{N}$, $f(n)$ is a real number, represented as an infinite decimal expansion that does *not* end with repeating 9's:

$$\begin{aligned}
f(0) &= i_0.d_{0,0}\,d_{0,1}\,d_{0,2}\cdots d_{0,n}\cdots \\
f(1) &= i_1.d_{1,0}\,d_{1,1}\,d_{1,2}\cdots d_{1,n}\cdots \\
f(2) &= i_2.d_{2,0}\,d_{2,1}\,d_{2,2}\cdots d_{2,n}\cdots \\
\vdots \quad &= \qquad\qquad \vdots \\
f(n) &= i_n.d_{n,0}d_{n,1}d_{n,2}\cdots d_{n,n}\cdots \\
\vdots \quad &= \qquad\qquad \vdots
\end{aligned}$$

where $i_0, i_1, \ldots \in \mathbb{Z}$ are the integer parts of the real numbers $f(0), f(1), \ldots$, and $\forall i \in \mathbb{N}, \forall j \in \mathbb{N}$, $d_{i,j} \in \{0,1,2,3,4,5,6,7,8,9\}$.

Now, let $r = 0.d_0d_1d_2\cdots d_n\cdots$, where $\forall i \in \mathbb{N}$,

$$d_i = \begin{cases} 1 & \text{if } d_{i,i} = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Then $r \in \mathbb{R}$. # r is an infinite decimal that does not end with repeating 9's

Then $\exists k \in \mathbb{N}, f(k) = r$. # f is onto

Since $f(k) = i_k.d_{k,0}d_{k,1}d_{k,2}\cdots d_{k,n}\cdots$ and $r = 0.d_0d_1d_2\cdots d_n\cdots$, this implies $i_k = 0$ and $\forall n \in \mathbb{N}$,

$$d_{k,n} = d_n = \begin{cases} 1 & \text{if } d_{n,n} = 0, \\ 0 & \text{otherwise.} \end{cases}$$

But then,

$$d_{k,k} = d_k = \begin{cases} 1 & \text{if } d_{k,k} = 0, \\ 0 & \text{otherwise,} \end{cases}$$

i.e., $d_{k,k} = 0 \Leftrightarrow d_{k,k} = 1$, a contradiction!

Then, by contradiction, $\mathbb{R}$ is not countable.

Take the time to study this proof a little...

You should notice the following key elements:

- The construction of a real number r such that $\forall n \in \mathbb{N}, f(n) \neq r$.
- The fact that this construction is carried out for an arbitrary function $f : \mathbb{N} \to \mathbb{R}$.

You should be able to use these ideas to write a direct proof that there does not exist any function $f : \mathbb{N} \to \mathbb{R}$ that is onto — thus proving directly that $\mathbb{R}$ is uncountable.

DIAGONALIZATION

The construction of real number r in the last proof is an example of DIAGONALIZATION. The reason for this name becomes obvious if we visualize how r is constructed: each digit d_i of r is defined to be different from at least one digit $d_{i,i}$ in the decimal expansion of the real number $f(i)$.

$$
\begin{array}{rcl}
r = & 0. & \boxed{d_0}\ \boxed{d_1}\ \boxed{d_2} \cdots \boxed{d_n} \cdots \\
f(0) = & i_0. & \boxed{d_{0,0}}\ d_{0,1}\ d_{0,2} \cdots d_{0,n} \cdots \\
f(1) = & i_1. & d_{1,0}\ \boxed{d_{1,1}}\ d_{1,2} \cdots d_{1,n} \cdots \\
f(2) = & i_2. & d_{2,0}\ d_{2,1}\ \boxed{d_{2,2}} \cdots d_{2,n} \cdots \\
\vdots & = & \qquad\qquad\ \vdots \\
f(n) = & i_n. & d_{n,0}\ d_{n,1}\ d_{n,2} \cdots \boxed{d_{n,n}} \cdots \\
\vdots & = & \qquad\qquad\ \vdots
\end{array}
$$

Because there are infinitely many digits in the decimal expansion of r (one for each natural number $n \in \mathbb{N}$), r is different from the real number $f(n)$ for every $n \in \mathbb{N}$.

Now, remember that we started this discussion in the context of the proof that function `halt` cannot be implemented in Python. Back then, I mentioned that this proof was an example of diagonalization — I will now explain why.

First, notice that `halt` takes two arguments: a one-argument Python function `f` and an input `i` for `f`. Clearly, there are infinitely many one-argument Python functions `f`, but what kind of infinity: countable or uncountable? To answer this question, we need a specific point of view: remember that every Python function can be written down as a text file, *i.e.*, a finite sequence of characters, from a fixed set of characters. To be specific, let's assume the character set is UTF-8, which contains 256 different characters — the exact details of the encoding used for the set of characters don't matter: what matters is that the set of possible characters is fixed, *i.e.*, the *same* finite set of characters can be used to write down every possible Python function. So, every Python function can be written down as a sequence of characters in UTF-8, and every such sequence of characters is really just an integer in disguise: just treat each character as a "digit" in base-256 notation.

If you think about it, what we've just done in the argument above is to define a function

$$g : \text{one-argument Python functions} \to \mathbb{N}$$

that is one-to-one: different Python functions get assigned to different numbers, because their source code must differ by at least one character. So by definition, the set of one-argument Python functions is countable — actually, the argument is more general than this: it shows that the set of *all* Python programs is countable.

By the definition of countability, this means that there is some other function

$$g' : \mathbb{N} \to \text{one-argument Python functions}$$

that is onto, *i.e.*, it is possible to *list* every one-argument Python function: $g'(0), g'(1), g'(2), \ldots$ — to make the notation easier to understand, we'll use subscripts $f_0 = g'(0), f_1 = g'(1), f_2 = g'(2), \ldots$ in the rest of the argument.

Keep in mind that each f_n is actually just a *string*: the source code for some one-argument Python function. So it makes sense to consider the function call $f_i(f_j)$ for $i, j \in \mathbb{N}$: we are simply calling the Python function whose source code is given by f_i with the string f_j as argument.

Now, consider the following table of behaviours, where we list one-argument Python functions down the left side and inputs for those functions across the top — our list only contains specific kinds of inputs (those that are listings for one-argument Python functions), and it is therefore not a complete list of all possible inputs. The entry at row f_i and column f_j states whether the call $f_i(f_j)$ halts or not (the values shown below are just an example).

	f_0	f_1	f_2 $\cdots$	f_n $\cdots$
f_0	halts	halts	loops $\cdots$	halts $\cdots$
f_1	loops	loops	loops $\cdots$	halts $\cdots$
f_2	loops	halts	halts $\cdots$	loops $\cdots$
$\vdots$			$\vdots$	
f_n	halts	halts	halts $\cdots$	halts $\cdots$
$\vdots$			$\vdots$	

When we assume that `halt` can be coded up in Python, this implies that every entry in this table can be computed by `halt`. Then, the function `confused` is created by a simple process of diagonalization over the table: just use `halt` to figure out the behaviour of $f_i(f_i)$ and make `confused` do the opposite. If `halt` exists, then so does `confused`, but `confused` cannot be any of the elements of the list $f_0, f_1, f_2, dotsc$ — by construction, its behaviour differs from each one of these functions. This is how we get a contradiction.

THINKING ABOUT SIZES AGAIN

There is something counter-intuitive about function `halt`: we can describe what the call `halt(f,i)` is supposed to return, even though the preceding argument shows that there is no way to implement function `halt` in Python (or in any other language, for that matter). This is different from every other Python function you've thought about: usually, you start with a vague, intuitive idea of what you want to accomplish, and you turn it into a working piece of Python code. But in this case, the process does not carry through: we can define the behaviour of function `halt`, but it cannot be implemented.

This shows that there are one-argument functions whose behaviour can be defined clearly, but that cannot be computed by any algorithm. But it also shows that *every* list of one-argument functions is incomplete. If you look at if again more carefully, you should see that the argument can be interpreted as a proof that the set of one-argument functions (meaning functions' behaviours) is uncountable.

Since we've already argued that the set of Python programs is countable, this means there are uncountably many functions that cannot be implemented in Python! Here is some standard terminology regarding these issues.

DEFINITION: A function $f : A \to B$ is COMPUTABLE if it can be implemented in Python, *i.e.*, if there exists a Python function `f` such that for all $a \in A$, `f(a)` returns the value of $f(a)$.

Functions for which this is not possible (*e.g.*, `halt`) are called NON-COMPUTABLE. Note that this definition applies only to well-defined mathematical functions $f : A \to B$ — those for which there is exactly one value $f(a)$ for every $a \in A$. The distinction between "computable" and "non-computable" has nothing to do with the *definition* of the function f. Rather, it distinguishes between functions whose values can be calculated by an algorithm, and those for which this is not possible.

6.5 REDUCTIONS

It's one thing to say that there are uncountably many non-computable functions, but are there any functions *that we would wish to compute* but that are not computable?

We've already seen one example: `halt(f,i)`. In this section, we'll explain how to use this result to show that other functions are non-computable.

Consider the function `initialized(f,v)`, which should return `True` when variable `v` is guaranteed to be initialized before its first use, whenever `f` is called (no matter what input is passed to `f`) — `initialize(f,v)` should return `False` if there is even one input `i` such that the call `f(i)` attempts to use the value of `v` before it has been initialized. Note that there is a subtle, but important, distinction to be made: we don't want `initialized(f,v)` to return `True` when there is just a *possibility* that variable `v` may be used before its

initialization in `f` — we want to know that this actually happens during the call `f(i)` for some input `i`. For example, consider the following functions:

```
def f1(x):
    return x + 1
    print y
```

```
def f2(x):
    return x + y + 1
```

While it is the case that `f1` contains a statement that uses variable `y` before it is initialized, this statement can never actually be executed — so `initialized(f1,y) == True`, vacuously. On the other hand, variable `y` is actually used before its initialization in `f2` — so `initialized(f2,y) == False`.

Claim 6.5: The function `initialized` is non-computable.

Proof:

For a contradiction, assume that `initialized` is computable, *i.e.*, it can be implemented as a Python function.

(We want to show that this assumption leads to a contradiction. More specifically in this case, we want to reach the contradiction that `halt` is computable.) Consider the following program.

```
def halt(f,i):
    def initialized(f,v):
        ...code for initialized goes here...

    # Code to scan the string f and figure out a variable name 'v'
    # that does not appear anywhere in f.
    def f_prime(i):
        f(i)
        print v

    return not initialized(f_prime,v)
```

If `f(i)` halts, then `f_prime(i)` will execute the statement `print v`, so `initialized(f_prime,v)` returns `False` and `halt(f,i)` returns `True`.

If `f(i)` does not halt, then `f_prime(i)` never executes `print v`, so `initialized(f_prime,v)` returns `True` and `halt(f,i)` returns `False`.

But there is no Python implementation of function `halt`, as we've already shown!

Hence, there is no Python implementation of function `initialized`.

Let's step through the preceding argument to better understand it.

- We want to prove that `initialized` is non computable (*i.e.*, that there is no Python code to compute it), based on the fact that `halt` is non-computable.

  We decide to use a proof by contradiction, so we begin by assuming that `initialized` *is* computable.

  Our goal is now to derive a contradiction. An obvious candidate is the statement: "`halt` is computable."

- To show that `halt` is computable means to show that there exists an algorithm to compute it. The easiest way to achieve this is to describe an explicit algorithm for `halt`.

  In other words, our goal is to find a way to compute `halt`, given a supposed algorithm for `initialized`.

- The trick to achieving this is contained in the definition of function `f_prime` in the argument: this function is valid Python code and it has been defined with the property that the call `f(i)` halts iff variable `v` is used without being initialized in `f_prime`.

  Moreover, this property does *not* depend on our knowledge of whether or not `f(i)` halts.

- Hence, we know that our implementation of `halt` will return the correct value, under the assumption that `initialized` has been implemented correctly.
- The contradiction follows immediately, which concludes the proof.

Note that the critical step in this argument — the one that requires the most creativity and insight — is to find a way to tie together the fact that "the call `f(i)` halts" with the fact that "variable `v` is always initialized before its use within function `f_prime`." Once we figure out how to achieve this, the structure of the rest of the proof is straightforward.

Chapter 6 Notes

[1]Yes, it would. And it turns out to be possible to give just such an argument, as you'll see.

[2]Yes, Python does allow such "nested" function definitions: it simply makes `'halt'`defined locally within `'confused'`,just like for variables.

[3]Notice this is just an application of Excluded Middle.

[4]Functions, conditionals, and loops. We used "nested functions," but that is not strictly necessary.

[5]$f(0) = 0/2 = 0 = (1-1)/2 = f(1)$.