# Chapter 0:
# Preface and Overview

Assume that all Greeks are Men. Assume also that all Men are mortal. It follows logically that all Greeks are mortal.

This deduction is remarkable in the sense that we can make it even without understanding anything about Greeks, Men, or mortality. The same deduction exactly can take the assumptions that all Greeks are fish and that all fish fly and conclude that all Greeks fly. As long as the assumptions are correct, so is the conclusion. If one or more of the assumptions is incorrect, then all bets are off and the conclusion need not hold. How are such "content free" deductions made? When is such a deduction valid? For example, assume that some Greeks are Men and that some Men are mortal; does it follow that some Greeks are mortal? No!

The field of Logic deals exactly with these types of deductions—those that do not require any specific knowledge of the real world, but rather take statements about the world and deduce new statements from them, new statements that must be true if the original ones are. Such deductions are the principle way by which we extend our knowledge beyond any facts that we directly observe. While in many fields of human endeavor logical deductions go hand in hand with other techniques of observing and understanding the actual facts of the world, in the field of Mathematics logical deductions serve as the sole paradigmatic foundation.

A crucial property of logical deduction is that it is **syntactic** rather than **semantic**. That is, the validity of a logical inference is completely determined by its language, its form, its syntax. Nothing about the actual meaning of the assumptions or conclusion, such as their truth or falsehood, is involved. The usefulness, however, of such deductions comes from the, perhaps surprising, fact that their conclusions do turn out to be true in the meaningful, semantic, sense. That is, whenever the assumptions are true, also the conclusion happens to be true—and this happens despite the fact the the deduction process itself was completely oblivious to said truth! Indeed, the clear separation between syntactic notions and semantic ones, as well as establishing the connections between them, are the core of the study of Logic. There are several different possible motivations for such study, and these different motivations influence the type of issues emphasized.

Philosophers will usually want to use Logic as their tool of the trade, and will mostly focus on the difficult process of translating between human questions and Logical Formulae[1]. These are tricky questions mostly due to the human part of this mismatch: human language is not completely precise, and to really understand the meaning of a sentence may require not only logical analysis but also linguistic analysis and even social understanding. For example, who exactly is included in the set of Greeks? When we assumed that they are all Men, does that include or exclude Women? The reader can surely see that without coming to grips with these thorny questions, one cannot assess

[1] "Formulae" is a plural form of "formula."

whether the assumptions are true and cannot benefit from the logical deduction that all Greeks are mortal.

Mathematicians also study Logic since it is their tool of the trade. Mathematicians usually apply Logic to precise mathematical statements, so they put less emphasis on the mismatch with the imprecise human language, but are rather focused on the exact rules of Logic and on exactly understanding the formalization, process, and power of Logic itself. Indeed, to understand the power of Logic is exactly to understand the limits of the basic paradigm of Mathematics and mathematical proofs, and thus the field of Mathematical Logic is sometimes called meta-Mathematics, mathematically studying *Mathematics* itself.

Computer Scientists use Logic as a tool of the trade in a somewhat different sense, often relying on logical formalisms to represent various computational abstractions. Thus, for example, a language to access databases (e.g., SQL) may be based on some logical formalism (e.g., Predicate Logic), and abstract computational search problems (e.g., NP problems) may be treated as finding assignments to logical formulae (e.g., SAT).

The approach of this book is to proceed towards the goal of Mathematicians who study Logic, using the tools of Computer Scientists, and in fact not those Computer Scientists who study Logic, but rather more applied Computer Scientists. Specifically, our main goal is to exactly formalize and understand the notions of a logical formula and a deductive logic proof, and to establish their relationship with mathematical truth. Our technique is to actually implement all these logical formulae and logical proofs as *bona fide* objects in a software implementation: you will actually be asked to implement, in the Python programming language, methods and functions that deal with Python objects like `Formula` and `Proof`. For example, in Chapter 2 you will be asked to implement a function `is_tautology(formula)` that determines if the given logical formula is a tautology, i.e., logically always true; while in Chapter 6 you will be asked to implement a function `proof_or_counterexample(formula)` that either returns a formal logical proof of the given formula—if it happens to be a tautology—or, otherwise, returns a counterexample that demonstrates that this formula is in fact *not* a tautology.

# 1 Our Final Destination: Gödel's Completeness Theorem

This book has a very clear end point to which everything leads: Gödel's Completeness Theorem. To understand it, let us first look at the two main syntactic objects that we will study and at their semantics. Our first focus of attention is the **Formula**, a formal representation of certain logical relations between basic simpler notions. For example a formalization of "All Men are mortal" in the form, say, '$\forall x[\text{Man}(x)\rightarrow\text{Mortal}(x)]$', and we will, of course, specify exact syntactic rules for such formulae. Now comes the semantics, that is, the notion of the meaning of such a formula. A formula may be true or false in a particular setting, depending on the specifics of the setting. Specifically, a formula gets a meaning only relative to a particular **model**, where this "model" must specify all the particulars of the setting. In our example, such particulars would include which $x$ in the "universe" are Men and which are mortal. Once such a model is given, then one can check whether a given formula is true in this model or not.

Our second focus of attention is the notion of a **Proof**. A proof again is a syntactic object: it has a set of **assumptions**, a **conclusion**, and the core of the proof is a list of

formulae that has to conform to certain specific rules ensuring that each formula in the list "follows" in some precise *syntactic* sense from previous ones. If such a formal proof exists, then we say that the conclusion is (syntactically) provable from the assumptions, which we denote by $\textit{assumptions} \vdash \textit{conclusion}$. Now, again, enter the semantics, which deal with the following question: is it the case that in *every* model in which all the assumptions are true, the conclusion is also true? (This question is only about the assumptions and the semantics, and is agnostic of the core of any proof.) If that happens to be true, then we say that the conclusion (semantically) follows from the assumptions, which we denote by $\textit{assumptions} \models \textit{conclusion}$. Gödel's theorem states the following:

**Theorem** (Gödel's Completeness)**.** *For any set of assumptions and any conclusion, it holds that "assumptions* $\vdash$ *conclusion" if and only if "assumptions* $\models$ *conclusion".*

This is a very remarkable theorem connecting two seemingly unrelated notions: the existence of certain long lists of formulae built according to some syntactic rules (these long lists are the syntactic proofs defined above), and the mathematical truth that whenever all assumptions are true, so invariably is the conclusion. On second thought, it does make sense that if something is syntactically provable then it is also semantically true: we will deliberately choose the syntactic rules of a proof to only allow true deductions. In fact, this is the whole point of Mathematics: in order to know that whenever we add two even numbers we get an even number, we do not need to check all possible (infinitely many!) pairs of even numbers, but rather it suffices to "prove" the rule that $\textit{even} + \textit{even}$ is *even* and the whole point is that a "proved" statement must be true (otherwise the concept of a proof would not have been of any use). The other direction, the fact that any mathematical truth can be proven, is much more surprising: we could have expected that the more possibilities we build into our proof system, the more mathematical truths it can prove. It is far from clear that any specific, finite, syntactic set of rules for forming proofs should suffice for proving *everything* that is true. And yet, for the simple syntactic set of logical rules that we will present, this is exactly what Gödel's theorem establishes.

One can view this as the final triumph of mathematical reasoning: our logical notion of proof completely suffices to establish any consequence of any list of assumptions. Given a set of axioms of, e.g., a mathematical **field** (or any other mathematical structure), anything that holds for all fields can actually be logically proven from the field axioms!

Unfortunately, shortly after proving this completeness theorem, Gödel turned his attention to the question of finding the "correct" set of axioms to capture all of Mathematics. What he desired was to find a simple set of axioms that suffices for proving or disproving any possible mathematical statement. We say "unfortunately" since Gödel failed in this second mission in a most spectacular way, showing that no such set of axioms exists: for every set of axioms there will remain mathematical statements that can neither be proved nor disproved! This is called Gödel's Incompleteness Theorem. Despite its name, this theorem does not in fact contradict the completeness theorem: it is still true that anything that (semantically) follows from a set of axioms is syntactically provable from it, but unfortunately there will always remain statements that neither they nor their negation follow from the set of axioms.

One can view Gödel's Incompleteness Theorem as the final defeat of mathematical reasoning: there will always remain questions beyond the reach of Mathematics. But this book—a first course in Mathematical Logic—focuses only on the triumph, i.e., on Gödel's Completeness Theorem, leaving the defeat, the Incompleteness Theorem, for a second course in Mathematical Logic.

# 2 The Pedagogical Approach

The mathematical content covered by this book is very standard for a first course in Mathematical Logic. Our pedagogical approach is, however, unique: we will "prove" everything by writing computer programs.

Let us motivate this unusual choice. We find that among academic courses in Mathematics, the introductory Mathematical Logic course stands out as having an unusual gap between student perceptions and our own evaluation of its content: while we (and, we think, most Mathematicians) view the mathematical content as rather easy, students seem to view it as very confusing relative to other Mathematics courses. While we view the conceptual message of the course as unusually beautiful, students often fail to see this beauty—even those that easily see the beauty of, say, Calculus or Algebra. We believe that the reason for this mismatch is the very large gap that exists between the very abstract point of view—proving things about proofs—and the very low-level technical proofs themselves. It is easy to get confused between the proofs that we are writing and the proofs that are our subjects of discussion. Indeed, when we say that we are "writing proofs to prove things about proofs", the first "proofs" and the second "proofs" actually mean two very different things even though most introductory Mathematical Logic courses use the same word for both. This turns out to become even more confusing as the "mechanics" of both the proof we are writing and the proof that we are discussing are somewhat cumbersome while the actual point that we are making by writing these proofs is something that we usually take for granted, so it is almost impossible to see the forest for the trees.

Computer Scientists are used to combining many "mechanical details" to a get a high-level abstract message (this is known as "programming"), and are also used to writing programs that handle objects that are as complex as themselves (such as compilers). A large part of Computer Science is exactly the discussion of how to handle such challenges both in terms of tools (debuggers, assemblers, compilers) and it terms of paradigms (interfaces, object-orientation, testing). So this book utilizes the tools of a Computer Scientist to achieve the pedagogical goal of teaching the mathematical basis of Logic.

We have been able to capture maybe 95% of the mathematical content of a standard first course in Mathematical Logic as programming tasks. These tasks capture the notions and procedures that are studied, and the solution to each of these programming tasks can be viewed as capturing the proof for some lemma or theorem. The reader that has actually implemented the associated function has in effect proved the lemma or theorem, a proof that has been verified for correctness (to some extent) once it has passed the extensive array of tests that we provide for the task. The pedagogical gain is that confusing notions and proofs become crystal clear once you have implemented them yourself. Indeed, in the above sentence "writing proofs to prove things about proofs", the first "proofs" becomes "code" and the second "proofs" becomes "Python objects of class `Proof`". Almost all the lemmas and theorems covered by a typical introductory course in Mathematical Logic are captured this way in this book. Essentially the only exceptions are theorems that consider "infinite objects" (e.g. an infinite set of formulae), which cannot be directly captured by a program that has to deal with finite objects. It turns out, however, that most of the mathematical content of even these infinitary proofs can be naturally captured by lemmas dealing with finite objects. What remains to be made in a purely mathematical way is just the core of the infinite argument, which is the remaining 5% or so that we indeed then lay out in the classical mathematical way.

# 3 How We Travel: Programs that Handle Logic

This book is centered around a sequence of programming projects in the Python programming language. We provide a file directory that contains a small amount of code that we have written, together with many skeletons of functions and methods that you will be asked to complete and an extensive array of tests that will verify that your implementation is correct. Each chapter of this book is organized around a sequence of tasks, each of which calls for completing the implementation of a certain function or method for which we have supplied the skeleton.

Let us take as an example Task 2 in Chapter 1. Chapter 1 deals with **propositional formulae**. You will handle such objects using code that appears in the Python file `propositions/syntax.py`, which contains a Python class `Formula` for holding a propositional formula as a tree-like data structure:[2]

propositions/syntax.py

```
class Formula:
    """An immutable propositional formula in tree representation.

    Attributes:
        root: the constant, atomic proposition, or operator at the root of the
            formula tree.
        first: the first operand to the root, if the root is a unary or binary
            operator.
        second: the second operand to the root, if the root is a binary
            operator.
    """
    root: str
    first: Optional[Formula]
    second: Optional[Formula]

    def __init__(self, root: str, first: Optional[Formula] = None,
                 second: Optional[Formula] = None) -> None:
        """Initializes a `Formula` from its root and root operands.

        Parameters:
            root: the root for the formula tree.
            first: the first operand to the root, if the root is a unary or
                binary operator.
            second: the second operand to the root, if the root is a binary
                operator.
        """
        if is_variable(root) or is_constant(root):
            assert first is None and second is None
            self.root = root
        elif is_unary(root):
            assert type(first) is Formula and second is None
            self.root, self.first = root, first
        else:
            assert is_binary(root) and type(first) is Formula and \
                   type(second) is Formula
            self.root, self.first, self.second = root, first, second
```

[2]The annotations following various colons signs, as well as following the `->` symbol, are called Python **type annotations** and specify the types of the variables/parameters that they follow, and respectively of the return values of the functions that they follow.

The main content of Chapter 1 is captured by asking you to implement various methods and functions related to objects of class `Formula`. Task 2 in Chapter 1, for example, asks you to implement the method `variables()` of this class, which returns a Python `set` of all named atomic propositions in the formula. The file `propositions/syntax.py` thus contains also the skeleton of this method:

propositions/syntax.py

```
class Formula:
        ⋮
    def variables(self) -> Set[str]:
        """Finds all atomic propositions (variables) in the current formula.

        Returns:
            A set of all atomic propositions used in the current formula.
        """
        # Task 1.2
```

To check that your implementation is correct, we also provide a corresponding test file, `propositions/syntax_test.py`, that contains the following test:

propositions/syntax_test.py

```
def test_variables(debug=False):
    for f, vs in [(Formula('T'), set()),
                  (Formula('x1234'), {'x1234'}),
                  (Formula('~',Formula('r')), {'r'}),
                  (Formula('->',Formula('x'), Formula('y')), {'x','y'}),
                      ⋮
                  (Formula(···), {···})]:
        if debug:
            print("Testing variables of", f)
        assert f.variables() == vs
```

All the tests of all tasks in Chapter 1 can be invoked by simply executing the Python file `test_ex1.py`, which we have also provided. If you run this file and get no assertion errors, then you have successfully (as far as we can check) solved all of the tasks in Chapter 1.

This chapter—Chapter 0—contains a single task, whose goal is to verify that you have successfully downloaded the code-base that comes with this book, and that your Python environment is correctly set up.

**Task 1.** Implement the missing code for the function `half(x)` from the file `prelim/prelim.py`, which halves an even integer. Here is the skeleton of this function as it appears in the file:

prelim/prelim.py

```
def half(x: int) -> int:
    """Halves the given even integer.

    Parameters:
        x: an even integer.

    Returns:
        A number `z` so that ``z+z=x``.
    """
    assert x%2 == 0
    # Task 0.1
```

The solution to Task 1 is very simple, of course (`return x//2`, or alternatively, `return int(x/2)`), but the point that we want you to verify is that you can execute the file `test_ex0.py` without getting any assertion errors, but only getting the expected verbose listing of what was tested:

```
$ python test_ex0.py
Testing half of 42
Testing half of 8
$
```

For comparison, executing the file `test_ex0.py` with a faulty implementation of Task 1 would raise an assertion error. For example, implementing Task 1 with, say, `return x//3`, would yield the following output:

```
$ python test_ex0.py
Testing half of 42
Traceback (most recent call last):
  File "test_ex0.py", line 13, in <module>
    test_task1(True)
  File "test_ex0.py", line 11, in test_task1
    test_half(debug)
  File "prelim/prelim_test.py", line 15, in test_half
    assert result + result == 42
AssertionError
$
```

and implementing Task 1 with, say, `return x/2` (which returns a `float` rather than an `int`), would yield the following output:

```
$ python test_ex0.py
Testing half of 42
Traceback (most recent call last):
  File "test_ex0.py", line 13, in <module>
    test_task1(True)
  File "test_ex0.py", line 11, in test_task1
    test_half(debug)
  File "prelim/prelim_test.py", line 14, in test_half
    assert type(result) is int
AssertionError
$
```

# 4 Our Roadmap

We conclude this chapter by giving a quick overview of our journey in this book. We study two logical formalisms: Chapters 1–6 deal with the limited **Propositional Logic**, while Chapters 7–12 move on to the fuller (first-order) **Predicate Logic**. In each of these two parts, we take a somewhat similar arc:

a. Define a syntax for logical formulae (Chapter 1 / Chapter 7),

b. define the semantics of said formulae (Chapter 2 / Chapter 7),

c. pause a bit in order to simplify things (Chapter 3 / Chapter 8),

d. define (syntactic) formal proofs (Chapter 4 / Chapter 9),

e. prove useful lemmas about said formal proofs (Chapter 5 / Chapters 10 and 11),

f. prove that any formula that is semantically true also has a syntactic formal proof (Chapter 6 / Chapter 12).

Of course, the results that we prove for the simpler Propositional Logic in the first part of this book, are then also used when dealing with Predicate Logic in the second part of the book. Here is a more specific chapter-by-chapter overview:

1. Chapter 1 defines a syntax for Propositional Logic and shows how to handle it.
2. Chapter 2 defines the notion of the semantics of a propositional formula, giving every formula a truth value in every given model.
3. Chapter 3 looks at the possible sets of logical operations allowed and discusses which such subsets suffice.
4. Chapter 4 introduces the notion of a formal deductive proof.
5. Chapter 5 starts analyzing the power of formal deductive proofs.
6. Chapter 6 brings us to the pinnacle of Part I, obtaining the "Tautology Theorem," which is a "completeness theorem" for Propositional Logic.
7. Chapter 7 starts our journey into Predicate Logic, introducing both its syntax and its semantics.
8. Chapter 8 is concerned with allowing some simplifications in our Predicate Logic, specifically getting rid of the notions of functions and of equality without weakening the expressive power of our formalism.
9. Chapter 9 introduces and formalizes the notion of a deductive proof of a formula in Predicate Logic.
10. Chapter 10 fixes a list of logical axioms and demonstrates their capabilities by applying them to several domains from syllogisms to mathematical structures to the foundations of mathematics, e.g., formalizing Russell's paradox about "the set of all sets that do not contains themselves."
11. Chapter 11 proves key results about the power of proofs in Predicate Logic.
12. Chapter 12 reaches the culmination of our journey by proving Gödel's Completeness Theorem. We also get, "for free," the "Compactness Theorem" of Predicate Logic.
13. Finally, Chapter 13 provides a "sneak peek" into a second course in Mathematical Logic, sketching a proof of Gödel's Incompleteness Theorem.