0 Introduction 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 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 a principal way by which we can 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 purely **syntactic** rather than **semantic**. That is, the validity of a logical deduction can be completely determined by 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, the conclusion also happens to be true – and this happens despite the fact that 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 usually use logic as a tool of the trade, and mostly focus on the difficult process of translating between natural human language and logical **formulas**.[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

[1] Another frequently used plural form of "formula," which you may encounter in many books, is "formulae." For simplicity, in this book we will stick with "formulas."

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? Without coming to grips with these thorny questions, one cannot assess whether the assumptions are true and cannot benefit from the logical deduction that all Greeks are mortal.

Mathematicians also study logic as a 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 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 formulas (e.g., SAT).

The approach of this book is to proceed toward 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 precisely 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 formulas 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 such as `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 returns either a formal logical proof of the given formula – if it happens to be a tautology – or else a counterexample that demonstrates that this formula is in fact *not* a tautology.

0.1 Our Final Destination: Gödel’s Completeness Theorem

This book has a very clear end point to which everything leads: Gödel’s completeness theorem, named after its discoverer, the Austrian (and later American) logician and mathematician Kurt Gödel. To understand it, let us first look at the two main syntactic objects that we will study and 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, ‘∀x[Man(x)→Mortal(x)]’ (we will, of course, specify exact syntactic rules for such formulas). Now comes the semantics, that is, the notion of truth 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 can be evaluated 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, it is determined 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 consists of a set of formulas called **assumptions**, an additional formula called **conclusion**, and the core of the proof is a list of formulas that has to conform to certain specific rules ensuring that each formula in the list "follows" in some precise *syntactic* sense from previous ones or from assumptions, and that the last formula in the list is the conclusion. 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 conclusion, and is agnostic of the core of any proof.) If that happens to be the case, then we say that the conclusion (semantically) follows from the assumptions, which we denote by $\textit{assumptions} \models \textit{conclusion}$. Gödel's **completeness theorem** states the following under certain conditions.

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

This is a very remarkable theorem connecting two seemingly unrelated notions: The existence of a certain long list of formulas built according to some syntactic rules (this long list is the syntactic proof just defined), 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 if the two numbers that we add up are even then the result is even as well, and the whole point is that our proof system is **sound**: 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, though, that any specific, finite, syntactic set of rules for forming proofs should suffice for proving, given any set of assumptions, *every* conclusion that follows from it. And yet, for the simple syntactic set of logical rules that we will present, this is exactly what Gödel's completeness 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 set 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 the properties of the natural numbers. What was desired at the time was to find for every branch of mathematics a simple set of axioms that suffices for proving or disproving any possible mathematical statement in that branch.[2] We say "unfortunately" since Gödel showed this to fail in a most spectacular way, showing that no such set of axioms exists even for the natural numbers: for every set of axioms there will remain mathematical statements about the natural numbers 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 such 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 any specific formalization 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.

0.2 Our Pedagogical Approach

The mathematical content covered by this book is quite 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 many 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 get a high-level abstract goal (this is known as "programming"), and are also used to writing programs that handle objects that are as complex as the programs themselves (such as compilers). A large part of computer science exactly concerns the discussion of how to handle such challenges both in terms of tools (debuggers, assemblers, compilers) and it terms of paradigms

[2] This desire, formulated by the German mathematician David Hilbert, was called "Hilbert's Program."

(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 who 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 earlier 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 formulas), which cannot be directly captured by a program that is constrained to dealing 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 non-programmatic 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.

0.3 How We Travel: Programs That Handle Logic

This book is centered around a sequence of programming projects in the Python programming language.[3] We provide a file directory that contains a small amount of code that we have already implemented, 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 (which also appears as a code snippet in the book). All of our code-base, including the already implemented parts of the code, the skeletons, and the tests, can be downloaded from the book website at `www.LogicThruPython.org`.

Let us take as an example Task 2 in Chapter 1. Chapter 1 deals with **propositional formulas**. You will handle such objects using code that appears in the Python file `propositions/syntax.py`, which already contains the constructor for a Python class `Formula` for holding a propositional formula as a tree-like data structure.[4]

[3] Specifically, the code snippets in this book have been tested with Python 3.7. Please refer to the book website at `www.LogicThruPython.org` for updated information regarding compatibility of newer Python versions with our code-base.

[4] The annotations following various colon 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.

propositions/syntax.py

```
class Formula:
    """An immutable propositional formula in tree representation, composed from
    variable names, and operators applied to them.

    Attributes:
        root: the constant, variable name, or operator at the root of the
            formula tree.
        first: the first operand of the root, if the root is a unary or binary
            operator.
        second: the second operand of 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):
        """Initializes a `Formula` from its root and root operands.

        Parameters:
            root: the root for the formula tree.
            first: the first operand for the root, if the root is a unary or
                binary operator.
            second: the second operand for 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 first is not None and second is None
            self.root, self.first = root, first
        else:
            assert is_binary(root)
            assert first is not None and second is not None
            self.root, self.first, self.second = root, first, second
```

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 variable names used in the formula. The file `propositions/syntax.py` thus already contains also the skeleton of this method.

propositions/syntax.py

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

        Returns:
            A set of all variable names 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`, which contains the following test:

```
propositions/syntax_test.py
def test_variables(debug=False):
    for formula, expected_variables 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', formula)
        assert formula.variables() == expected_variables
```

We encourage you to always browse through the examples within the test code before starting to implement the task, to make sure that you fully understand any possible nuances in the specifications of the task.

All the tests of all tasks in Chapter 1 can be invoked by simply executing the Python file `test_chapter01.py`, which we also provide. The code for testing the optional tasks of Chapter 1 is commented out in that file, so if you choose to implement any of these tasks (which is not required in order to be able to implement any of the non-optional tasks that follow them), simply uncomment the corresponding line(s) in that file. 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 our code base from the book website at `www.LogicThruPython.org`, and that your Python environment is correctly set up.

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

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

    Parameters:
        x: even integer to halve.

    Returns:
        An integer `z` such 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_chapter00.py` without getting any assertion errors, but only getting the expected verbose listing of what was tested.

```
$ python test_chapter00.py
Testing half of 42
Testing half of 8
$
```

For comparison, executing the file `test_chapter00.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_chapter00.py
Testing half of 42
Traceback (most recent call last):
  File "test_chapter00.py", line 13, in <module>
    test_task1(True)
  File "test_chapter00.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_chapter00.py
Testing half of 42
Traceback (most recent call last):
  File "test_chapter00.py", line 13, in <module>
    test_task1(True)
  File "test_chapter00.py", line 11, in test_task1
    test_half(debug)
  File "prelim/prelim_test.py", line 14, in test_half
    assert isinstance(result, int)
AssertionError
$
```

0.4 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 of the book, we take a somewhat similar arc:

a. Define a syntax for logical formulas (Chapter 1/Chapter 7).
b. Define the semantics of said formulas (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 Part I of this book are then also used when dealing with predicate logic in Part II 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 of this book, obtaining the "tautology theorem," which is the mathematical heart of the completeness theorem for propositional logic (which we will indeed derive from it), and is also a key result that we will use in Part II of this book when proving the completeness theorem for predicate 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 set of logical axioms and demonstrates their capabilities by applying them to several domains from syllogisms, through 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.

Part I

Propositional Logic

1 Propositional Logic Syntax

In this chapter we present a formal **syntax** for formalizing statements within logic. Consider the following example of a natural language sentence that has some logical structure: "If it rains on Monday then we will either hand out umbrellas or rent a bus." This sentence is composed of three basic **propositions**, each of which may potentially be either true or false: p1="it rains on Monday", p2="we will hand out umbrellas", and p3="we will rent a bus". We can interpret this English-language sentence as logically connecting these three propositions as follows: "p1 implies (p2 or p3)", which we will write as '(p1→(p2|p3))'.

Our goal in this chapter is to formally define a language for capturing these types of sentences. The motivation for defining this language is that it will allow us to precisely and formally analyze their *implications*. For example, we should be able to formally deduce from this sentence that if we neither handed out umbrellas nor rented a bus, then it did not rain on Monday. We purposefully postpone to Chapter 2 a discussion of **semantics**, of the meaning, that we assign to sentences in our language, and focus in this chapter only on the **syntax**, i.e., on the rules of grammar for forming sentences.

1.1 Propositional Formulas

Our language for Part I of this book is called **propositional logic**. While there are various variants of the exact rules of this language (allowing for various logical operators or for various rules about whether and when parentheses may be dropped), the exact variant used is not very important, but rather the whole point is to fix a single specific set of rules and stick with it. Essentially everything that we say about this specific variant will hold with only very minor modifications for other variants as well. Here is the formal definition with which we will stick.

DEFINITION 1.1 (Propositional Formula) The following strings are (valid[1]) **propositional formulas**:

- A **variable name**: a letter in 'p'… 'z', optionally followed by a sequence of digits. For example, 'p', 'y12', or 'z035'.
- 'T'.
- 'F'.
- A **negation** '~ϕ', where ϕ is a (valid) propositional formula.

[1] What we call **valid** formulas are often called **well-formed** formulas in other textbooks.

- '$(\phi\&\psi)$' where each of ϕ and ψ is a propositional formula.
- '$(\phi|\psi)$' where each of ϕ and ψ is a propositional formula.
- '$(\phi\rightarrow\psi)$' where each of ϕ and ψ is a propositional formula.

These are the only (valid) propositional formulas. For example, '~((~x&(p007|x))→F)' is a propositional formula.

This definition is **syntactic**: it specifies which **strings**, that is, finite sequences of characters, are valid propositional formulas and which are not, by describing the rules through which such strings can be formed. (Again, we have deliberately not yet assigned any interpretation to such strings, but the reader will surely guess that the **constants** 'T' and 'F' stand for *True* and *False*, respectively, that the **unary** (operating on one **subformula**) **operator** '~' stands for *Not*, and that the **binary** (operating on two **subformulas**) **operators** '&', '|', and '→' stand for *And*, *Or*, and *Implies*, respectively.) We remark that in many logic textbooks, the symbol '¬' (**negation**) is used instead of '~', the symbol '∧' (**conjunction**) is used instead of '&', and the symbol '∨' (**disjunction**) is used instead of '|'.

Our choice of symbols in this book was indeed influenced by which symbols are easy to type on a computer. For your convenience, the file `propositions/syntax.py` defines functions for identifying strings that contain the various **tokens**, or basic building blocks, allowed in propositional formulas.[2] The symbol '→' is not a standard character, so in Python code we will represent it using the two-character sequence `'->'`.

propositions/syntax.py

```
def is_variable(string: str) -> bool:
    """Checks if the given string is a variable name.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a variable name, ``False`` otherwise.
    """
    return string[0] >= 'p' and string[0] <= 'z' and \
           (len(string) == 1 or string[1:].isdecimal())

def is_constant(string: str) -> bool:
    """Checks if the given string is a constant.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a constant, ``False`` otherwise.
```

[2] The decorator that precedes the definition of each of these functions in the code that you are given **memoizes** the function, so that if any of these functions is called more than once with the same argument, the previous return value for that argument is simply returned again instead of being recalculated. This has no effect on code correctness since running these functions has no side effects, and their return values depend only on their arguments and are immutable, but this does speed-up the execution of your code. It may seem silly to perform such optimizations with such short functions, but this will in fact dramatically speed-up your code in later chapters, when such functions will be called many many times from within various recursions. We use this decorator throughout the code that you are given in various places where there are speed improvements to be gained.