1.1 The Different Types of Data

Intro

- *Data type* – a way of categorizing data
- Data type conveys:
    - The allowed *values* for a piece of data
    - The allowed *operations* we can perform on a piece of data

Numeric Data

- A *natural number* is a value from the set {0, 1, 2, …}
    - Denoted by symbol $\mathbb{N}$
    - In computer science, 0 is a natural number
- An *integer* is a value from the set {…, -2, -1, 0, 1, 2, …}
    - Denoted by symbol $\mathbb{Z}$
- A *rational number* is a value from the set $\{\frac{p}{q} \mid p, q \in \mathbb{Z} \; and \; q \neq 0\}$
    - Denoted by symbol $\mathbb{Q}$
- An *irrational number* is a number with an infinite and non-repeating decimal expansion
    - i.e. pi, e
    - Denoted by symbol $\overline{\mathbb{Q}}$
- A *real number* is either a rational or irrational number
- Denoted by symbol $\mathbb{R}$

Operations on Numeric Data

- Standard arithmetic operations
- Standard comparisons for equality and inequality
- *Modulo operator* that produces the remainder when one integer is divided by another
    - Denoted by symbol %
    - *a*%*b* means “the remainder when *a* is divided by *b*”

Boolean Data

- A *Boolean* is value from the set {True, False}
    - Yes/No question

Operations on Boolean Data

- Booleans can be combined using *logical operators*
- Three most common logical operators:
    - *Not* – reverses the value of a Boolean
        - i.e. “not True” is False

- *And* – takes two Boolean values and produces True when both of the values are True, and False otherwise
  - i.e. "True and False" is False, "True and True" is True
- *Or* – takes two Boolean values and produces True when at least one of the values is True, and False otherwise
  - i.e. "True or False" is True, "False or False" is False

Textual Data

- A *string* is an ordered sequence of characters, it's used to represent text

Writing Textual Data

- Strings are surrounded by single quotes to differentiate them from surrounding text
  - e.g. 'David'
- *Empty string* – a string that has zero characters
  - Denoted by '' or ϵ

Operations on Textual Data

- $|s|$
  - *String length/size*
  - Returns the number of characters in s
- $s_1 = s_2$
  - *String equality*
  - Returns whether s_1 and s_2 have the same characters, in the same order
- $s + t$
  - *String concatenation*
  - Returns a new string consisting of the characters of s followed by the characters of t
    - If s_1 represents the string 'Hello' and s_2 represents the string 'Goodbye', then $s_1 + s_2$ is the string 'HelloGoodbye'
- $s[i]$
  - *String indexing*
  - Returns the i-th character of s, where indexing starts at 0
    - $s[0]$ returns the first character of s, $s[1]$ returns the second, etc.

Set Data (Unordered Distinct Values)

- *Set* – an unordered collection of zero or more distinct values, called its *elements*
  - i.e. the set of all people in Toronto; the set of words of the English language; etc.

Writing Sets

- Two different ways:
  - Writing each element of the set within the braces, separated by commas
    - i.e. {1, 2, 3} or {'hi', 'bye'}
  - Using *set builder notation*, in which we define the form of elements of a set using variables
    - i.e. $\{\frac{p}{q} \mid p, q \in \mathbb{Z} \; and \; q \neq 0\}$
- *Empty set* – a set having zero elements
  - Denoted by {} or ∅

Operations on Set Data

- $|A|$
  - Returns the *size* of set *A*
    - i.e. the number of elements in *A*
- $x \in A$
  - Returns True when *x* is an element of *A*
- $y \notin A$
  - Returns True when *y* is *not* an element of *A*
- $A \subseteq B$
  - Returns True when every element of *A* is also in *B*
  - *A* is a *subset* of *B*
  - A set *A* is a subset of itself, and the empty set is a subset of every set
- *A* = *B*
  - Returns True when *A* and *B* contain the exact same elements
- $A \cup B$
  - The *union* of *A* and *B*
  - Returns the set consisting of all elements that occur in *A*, in *B*, or in both
  - $A \cup B = \{x \mid x \in A \; or \; x \in B\}$
- $A \cap B$
  - The *intersection* of *A* and *B*
  - Returns the set consisting of all elements that occur in both *A* and *B*
  - $A \cap B = \{x \mid x \in A \; and \; x \in B\}$
- $A \setminus B$
  - The *difference* of *A* and *B*
  - Returns the set consisting of all elements that are in *A* but that are not in *B*
  - $A \setminus B = \{x \mid x \in A \; and \; x \notin B\}$
- $A \times B$
  - The *(Cartesian) product* of *A* and *B*

    - Returns the set consisting of all *pairs* (a, b) where *a* is an element of *A* and *b* is an element of *B*
    - $A \times B = \{(x, y) \mid x \in A \; and \; y \in B\}$
- P(*A*)
    - The *power set* of *A*
    - Returns the set consisting of all subsets of *A*
    - i.e. if *A* = {1, 2, 3}, then P(*A*) = {∅, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}
    - $P(A) = \{S \mid S \subseteq A\}$

List Data (Ordered Values)

- *List* – an ordered collection of zero or more (possibly duplicated) values, called its elements
- Used instead of set when the elements of the collection should be in a specified order, or if it may contain duplicates
    - i.e. the list of all people in Toronto, ordered by age; the list of words of the English language, ordered alphabetically; etc.

Writing Lists

- Written with square brackets enclosing zero or more values separated by commas
    - i.e. [1, 2, 3]
- *Empty list* – a list having zero elements
    - Denoted by []

Operations on List Data

- |*A*|
    - Returns the *size* of *A*
        - i.e. the number of elements in *A* (counting all duplicates)
- $x \in A$
    - same meaning as for sets
- *A* = *B*
    - *A* and *B* have the same elements in the same order
- *A*[*i*]
    - *List indexing*
    - Returns the *i*-th element of *A*, where the indexing starts at 0
        - i.e. *A*[0] returns the first element of *A*, *A*[1] returns the second, etc.
- *A* + *B*
    - *List concatenation*
    - Returns a new list consisting of the elements of *A* followed by the elements of *B*

        - i.e. [1, 2, 3] + [2, 4, 6] = [1, 2, 3, 2, 4, 6]
    - Similar to set union, but duplicates are kept, and order is preserved

Mapping Data

- *Mapping* – an unordered collection of pairs of values
- Each pair consists of a *key* and an associated *value*
- The keys must be unique, but the values can be duplicated
- A key cannot exist without a corresponding value
- Used to represent associations between two collections of data
    - i.e. a mapping from the name of a country to its GDP; a mapping from student number to name; etc.

Writing Mappings

- Curly braces are used to represent a mapping
    - Similar to sets as both mappings and sets are unordered and both have a uniqueness constraint (a set's elements; a mapping's keys)
- Each key-value pair in a mapping is written using a colon, with the key on the left side of the colon and its associated value on the right
- i.e. {'fries' : 5.99, 'steak' : 25.99, 'soup' : 8.99}

Operations on Mappings

- $|M|$
    - Returns the *size* of the mapping *M*
        - i.e. the number of key-value pairs in *M*
- $M = N$
    - Returns whether two mappings are equal
        - i.e. when they contain exactly the same key-value pairs
- $k \in M$
    - Returns whether *k* is a *key* contained in the mapping *M*
- $M[k]$
    - When *k* is a key in *M*, this operation returns the value that corresponds to *k* in the mapping *M*

Extra

- Images can be represented as a list of integers
    - Each element in the list corresponds to a dot called *pixel* on the screen
    - For each dot, three integer values are used to represent three colour channels: red, green, and blue

    - We can add these channels together to get a very wide range of colours
        - Called the RGB colour model

1.2 Introducing the Python Programming Language

What is a Programming Language?

- A *programming language* is a way of communicating a set of instructions to a computer
- A *program* is the text of instructions we wish to instruct the computer to execute
    - We call this text program *code* to differentiate it from other forms of text
- Two key properties of the language:
    - *Syntax* – rules governing what constitutes a valid program in the language
        - Specifies format of instructions
    - *Semantics* – rules governing the *meaning* of different instructions in the language
        - Specifies what the computer should do for each instruction

The Python Programming Language

- *Python interpreter* takes programs written in the Python language and execute the instructions
- Two ways of writing code in Python to be understood by the interpreter
    - Write full programs in Python, saving them as text files
        - Python programs use the .py file extension
        - Writing the instructions, and then run them with the interpreter
    - Run the Python interpreter in an interactive mode, which we call the *Python console* or *Python shell*
        - We can write small fragments of Python code and ask the Python interpreter to execute each fragment one at a time
        - Useful for experimenting and exploring the language, as we get feedback after every single instruction
        - Drawback: interactions with the interpreter in the Python console are ephemeral, lost every time you restart the console
- In the course, we will use the Python console to learn about and experiment with the Python language, and write full programs in .py files

1.3 Representing Data in Python

The Python Console

- When we start the Python console, we see <<<

    - The text <<< is the Python *prompt*
        - The console is "prompting" us to type in some Python code to execute
- *Expression* – a piece of Python code that produces a value
- *Evaluation* – the act of calculating the value of an expression
- Expression 4 + 5 is formed from 2 smaller expressions – numbers 4 and 5
- *Literal* – simplest kind of Python expression
    - The piece of code that represents the exact value as written
    - i.e. 4 is an integer literal literally representing the number 4

<u>Numeric Data in Python (int, float)</u>

- Two data types for representing numeric data: *int* and *float*
- *int* – integer
    - int literal is simply an integer
    - i.e. 110; -3421
- 2 ** 5: 2 to the power of 5
- *Comment* – code ignored by the Python interpreter, begin comment with "#"
- Python follows the "BEDMAS" rule
- Python always produce an int value for addition, subtraction, multiplication, and exponentiation
- Python has 2 different forms of division
- *Floor division/Integer division* – operator //, x / y rounded down to the nearest integer
    - i.e. 6 // 2 outputs 3; 15 // 2 outputs 7; -15 // 2 outputs -8
- *Division* – operator /, returns a *float* type
- *float* – represents arbitrary real numbers
    - float literal is written as a sequence of digits followed by a decimal point (.), then another sequence of digits
    - i.e. 2.5; .123; 1000.00000001
- 2 ** 0.5 calculates the square root of 2, however, it is only an approximation of sqrt2, which does not equal it
    - Since sqrt2 is an irrational number, its decimal expansion cannot be represented in any finite amount of computer memory
- float cannot always represent real numbers exactly
- 3 vs. 3.0
    - 6 // 2 outputs 3; 6 / 2 outputs 3.0
    - When x and y are ints,
        - x // y *always* evaluates to an int
        - x / y *always* evaluates to a float
- Mixing ints and floats

    - For two ints x and y, x + y, x – y, x * y, x // y, and x ** y all produce ints; x / y always produces a float
    - For two floats, all six of the operations above produce a float
    - An arithmetic operation that is given one int and one float always produces a float
    - Even in long expressions where only 1 value is a float, the whole expression will evaluate to a float
- Operation description

a + *b*	Returns the sum of *a* and *b*
a – *b*	Returns the result of subtraction of *b* from *a*
a * b	Returns the result of multiplying *a* by *b*
a / *b*	Returns the result of dividing *a* by *b*
a % *b*	Returns the remainder when *a* is divided by *b*
a ** *b*	Returns the result of *a* being raised to the power of *b*
a // *b*	Returns the floored division *a* / *b*

- Comparison operators

a == *b*	Returns whether *a* and *b* are equal
a != *b*	Returns whether *a* and *b* are *not* equal (opposite of ==)
a > *b*	Returns whether *a* is less than the value of *b*
a < *b*	Returns whether *a* is less than the value of *b*
a >= *b*	Returns whether *a* is greater than or equal to *b*
a <= *b*	Returns whether *a* is less than or equal to the value of *b*

- Python can recognize when the values of ints and floats represent the same number
    - i.e. 4 == 4.0 outputs True

<u>Boolean Data in Python (bool)</u>

- Two literal values of type *bool*: True and False
- 3 boolean operators we can perform on boolean values:
    - not
    - and
    - or
    - i.e. not True outputs False; True and True outputs True; True and False outputs False; False or True outputs True; False or False operates False
- The *or* operator in Python is *inclusive*, which produces True when both of its operand expressions are True
- Boolean operator expressions can be combined with itself or the arithmetic comparison operators
    - i.e. True and (False or True) outputs True; (3 == 4) or (5 < 10) outputs False

Textual Data in Python (str)

- *str* – short for "string"
- A str literal is a sequence of characters surrounded by single-quotes (')
    - i.e. 'Foundations of Computer Science I'
- We can compare string using ==
    - i.e. 'David' == 'David' outputs True; 'David' == 'david' outputs False
- *String indexing* – extracting a single character from a string
    - Starts at 0 – s[0] is the *first* character in s
    - i.e. 'David'[0] outputs 'D'; 'David'[3] outputs 'I'
- *Concatenation* – operator +, combines strings
    - i.e. 'One string' + 'to rule them all.' outputs 'One stringto rule them all'; "One string' + ' to rule them all.' outputs 'One string to rule them all'
- *String repetition* – operator *, repeats string
    - i.e. 'David' * 3 outputs 'DavidDavidDavid'
- The above string operations can be nested within each other
    - i.e. ('David' + 'Mario') * 3 outputs 'DavidMarioDavidMarioDavidMario'

Set Data in Python (set)

- Python uses the *set* data type to store set data
- set literal begins with a { and ends with a }, and each element of the list is written inside the braces, separated from each other by commas
    - i.e. {1, 2, 3} is a set of ints; {1, 2.0, 'three'} is a set of elements of mixed types
- Sets can be compared for equality using ==
    - Order does not matter in sets
    - i.e. {1, 2, 3} == {3, 1, 2} outputs True
- *in* – equivalent to $\in$
    - i.e. 1 in {1, 2, 3} outputs True; 10 in {1, 2, 3} outputs False
- *not in* – equivalent to $\notin$
    - i.e. 1 not in {1, 2, 3} outputs False; 10 not in {1, 2, 3} outputs True

List Data in Python (list, tuple)

- Two different data types to store list data: *list* and *tuple*
- list literals are written the same way as set literals, except using square brackets [] instead of curly braces {}
- Lists support the same operations for strings and sets
    - [1, 2, 3] == [1, 2, 3] outputs True; [1, 2, 3] == [3, 2, 1] outputs False
    - (['David', 'Mario'])[0] outputs 'David'

    - ['David', 'Mario'] + ['Jacqueline', 'Diane'] outputs ['David', 'Mario', 'Jacqueline', 'Diane']
    - 1 in [1, 2, 3] outputs True
- tuple literals are written using regular parentheses ()
    - The above examples for list work the same way for tuples
    - () instead of []

Mapping Data in Python (dict)

- *dict* – short for "dictionary", data type for mapping data
- dict literals are similar to sets, with each key-value pair separated by a colon
    - i.e. {'fries': 5.99, 'steak': 25.99, 'soup': 8.99}
        - In this dictionary, the keys are strings, the values are floats
- The literal {} represents an empty dictionary (not an empty set)
    - Python has no literal to represent an empty set
- Dictionaries support equality comparison using ==
- They also support key lookup using the same syntax as string and list indexing
    - i.e. ({'fries': 5.99, 'steak': 25.99, 'soup': 8.99})['fries'] outputs 5.99
- *in* – checking whether a key is present in dictionary
    - i.e. 'fries' in {'fries': 5.99, 'steak': 25.99, 'soup': 8.99} outputs True

1.4 Storing Data in Variables

Variables

- *Variable* – a piece of code that *refers* to a value
- We create variables in Python using the syntax:
    - <variable> = <expression>
        - Called an *assignment statement*
        - When we execute an assignment statement, it doesn't produce a value – it instead defines a variable
- *Expression* – a piece of Python code that is evaluated to produce a value
- Python executes an assignment statement in 2 steps:
    1. The expression on the right side of the = is evaluated, producing a value
    2. That value is bound to the variable on the left side
- After the assignment statement is executed, the variable may be used to refer to the value

Choosing Good Variable Names

- To remember the purpose for each variable

- Rules for choosing variable names:
    - Use only lowercase letters, digits, and underscores
        - i.e. distance1 (not Distance1)
    - Write each word in lowercase and separate them with an underscore
        - i.e. total_distance = distance1 _ distance2 (not totaldistance or totalDistance)
    - Avoid single-letter variable names and non-standard acronyms/abbreviations
        - i.e. total_distance (not td)
        - A second person might not understand what td stands for

The Value-Based Python Memory Model

- *Memory model* – a structured way of representing variables and data in a program
    - *Memory* refers to computer memory
    - Uses a table to represent the associations between variables and their associated values
    - i.e.

Variable	*Value*
distance1	2.265409645965010650498645056
distance2	10.0

1.5 Build Up Data with Comprehensions

From Set Builder Notation to Set Comprehensions

- *Set builder notation* – a concise way of defining a mathematical set
    - S = {1, 2, 3, 4, 5}
    - A set of squares of the elements of S: $\{x^2 \mid x \in S\}$
- *Set comprehension* – set builder notation translated into Python
    - Syntax: {<expr> for <variable> in <collection>}
        - expr – expression
- Set comprehension is evaluated by taking the <expr> and evaluating it once for each value in <collection> assigned to the <variable>
    - Analogous to the set builder notation, using *for* instead of | and *in* instead of ∈
- Example
    - numbers = {1, 2, 3, 4, 5}
    - {x ** 2 for x in numbers}
    - Output: {1, 4, 9, 16, 25}
        - {x ** 2 for x in numbers} == {1 ** 2, 2 ** 2, 3 ** 2, 4 ** 2, 5 ** 2}
            - Replacing x with 1, 2, 3, 4, and 5

- We can use set comprehensions with a Python list as well
    - i.e. {x ** 2 for x in [1, 2, 3, 4, 5]} outputs {1, 4, 9, 16, 25}
- Set comprehensions can be used with any "collection" data type in Python

List and Dictionary Comprehensions

- *List comprehension* – similar to a set comprehension, but used square brackets [] instead of curly braces {}
    - Syntax: [<expr> for <variable> in <collection>]
        - <collection> can be a set or a list
    - i.e. [x + 4 for x in {10, 20, 30}] outputs [14, 24, 34]; [x * 3 for x in [100, 200, 300]] outputs [300, 600, 900]
- Do *not* assume a particular ordering of the elements when a list comprehension generates elements from a set
- *Dictionary comprehension* – similar to a set comprehension, but specifies both an expression to generate keys and an expression to generate their associated values
    - Syntax: {<key_expr> : <value_expr> for <variable> in <collections>}
    - Example of a dictionary comprehension that creates a "table of values" for the function f(x) = x^2 + 1:
        - {x : x ** 2 + 1 for x in {1, 2, 3, 4, 5}} outputs {1: 2, 2: 5, 3: 10, 4: 17, 5: 26}

Comprehensions with Multiple Variables

- *Cartesian product* of two sets:
    - $A \times B = \{(x, y) \mid x \in A \; and \; y \in B\}$
        - The expression (x, y) is evaluated once for every possible combination of elements x of A and elements y of B
- We can specify additional variables in a comprehension by adding extra *for <variable> in <collection>* clauses to the comprehension
    - nums1 = {1, 2, 3}
    - nums2= {10, 20, 30}
    - {(x, y) for x in nums1 for y in nums2}
    - Output: {(3, 30), (2, 20), (2, 10), (1, 30), (3, 20), (1, 20), (3, 10), (1, 10), (2, 30)}
        - Sets are unordered
- If we have a comprehension with clauses for v1 in collection1, for v2 in collection2, etc., then the comprehension's inner expression is evaluated *once for each combination of values for the variables*

2.1 Python's Built-In Functions

Functions in Mathematics

- Let *A* and *B* be sets, a *function f*: *A* → *B* is a mapping from elements in *A* to elements in *B*
    - *A* is called the *domain* of the function
    - *B* is called the *codomain* of the function
- Functions can have more than 1 input
    - $f: A_1 \times A_2 \times \ldots \times A_k \rightarrow B$
    - Functions that take one, two, and three inputs are *unary*, *binary*, and *ternary* functions, respectively
        - i.e. the addition operator + is a *binary* function ($\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$) that takes 2 real numbers and returns their sum

Python's Built-In Functions in Python

- *Built-in functions* – functions we can use to perform additional operations defined by Python
    - They are made automatically available to us anywhere in a Python program
- *Function call* – a Python expression that uses a funciton to operate on a given input
    - Syntax: <function_name> (<argument>, <argument>, …)
    - i.e. abs(-10) outputs 10
- *Arguments* – the input expresssions in a function call expression
    - i.e. the -10 above
- When we evaluate a function call, the arguments are *passed* to the function
    - i.e. -10 is passed to abs
- When a function call produces a value, we say that the functin call *returns* a value, and refer to this value as the *return value* of the function call expression
    - i.e. 10
- Examples of built-in functions
    - *len* – takes a string or a collection data type (e.g. set, list) and returns the size of its input
        - i.e. len({10, 20, 30}) outputs 3
    - *sum* – takes a collection of numbers (e.g. a set or list whose elements are all numbers) and returns the sum of the numbers
        - i.e. sum({10, 20, 30}) outputs 60
    - *sorted* – takes a collection and returns a listy that contains the same elemenmts as the input collection, sorted in ascending order
        - i.e. sorted({10, 3, 20, -4}) outputs [-4, 3, 10, 20]
    - *max* – when it is called with two or more numeric inputs, returns the largest one; when it is called with a collection of numbers, returns the largest number in the collection

        - i.e. max(2, 3) outputs 3; max({2, 3}) outputs 3
    - *range* – takes 2 integers *start* and *stop* and returns a value representing a range of consecutive numbers between *start* and *stop* – 1, inclusive
        - i.e. range(5, 10) represents 5, 6, 7, 8, 9; range(10, 5) represents an *empty* sequence
- One special function: type
    - Takes *any* Python value and returns its type
    - type(3) outputs <class 'int'>
    - type(3.0) outputs <class 'float'>
    - type('David') outputs <class 'str'>
    - type([1, 2, 3]) outputs <class 'list'>
    - type({'a': 1, 'b': 2}) outputs <class 'dict'>
- Nested function calls
    - We can write function calls within each other
        - i.e. max(abs(-100), 15, 3*20) outputs 100
    - Too much nesting can make Python expressions difficult to read and understand
    - It is better to break down a complex series of function calls into intermediate steps using variables

<u>Methods: Functions Belonging to Data Types</u>

- The built-in functions above can all be given arguments of at least two different data types
- *Method* – a function that is defined as part of a data type
    - All methods are functions but not vice versa
        - i.e. the built-in functions are not methods
- *Top-level functions* – functions that are not methods
- *str.lower* – takes a string and returns a new string with all uppercase letters turned into lowercase
    - i.e. str.lower('David') outputs 'david'
- To call a method, we refer to it by first specifying the name of the data type it belongs to (i.e. str), followed by a period (.), and then the name of the method (lower)
- *str.split* – splits a string into words
    - i.e. str.split('David wuz hear') outputs ['David', 'wuz', hear']
- *set.union* – performs the setr union operation
    - i.e. set.union({1, 2, 3}, {2, 10, 20}) outputs {1, 2, 3, 20, 10}
- *list.count* – counts the number of times a value appears in a list
    - i.e. list.count([1, 2, 3, 1, 2, 4, 2], 2) outputs 3

2.2 Defining Our Own Functions

Defining a Python Function

- A "squaring" function in Python
    - def square(x: float) -> float:
        """Return x squared.

        >>> square(3.0)
        9.0
        >>> square(2.5)
        6.25
        """
        return x ** 2
- *Function header* – the first line, def square(x: float) -> float:
    - Conveys the following pieces of information:
        - The function's name (square)
        - The number and type of arguments the function expects
            - *Parameter* – a variable in a function definition that refers to an argument when the function is called
            - The function has one parameter with name x and type float
        - The function's *return type*
            - The type following the -> (float)
    - Syntax for a function header for a unary function:
        - def <function_name> (<parameter_name>: <parameter_type>) -> <return_type>:
        - We choose the name square rather than f as the function name
        - We use data types to specify the function domain and codomain
            - x: float specifies that the parameter x must be a float value
            - -> float specifies that this function always return a float value
        - *Type contract* – the domain-codomain restriction in an analogous way to $f: \mathbb{R} \to \mathbb{R}$
            - float -> float
- *Function docstring* – the next lines that start and end with triple-quotes (""")
    - Another way of writing a comment in Python
    - Text that is meant to be read by humans, but not executed as Python code
    - Goal: to communicate what the function does
    - First part of the docstring, Return x squared, is an English description of the function

    - Second part of the docstring looks like Python code
        - The first example: "when you type square(3.0) into the Python console, 9.0 is returned
        - The second example : "when you type square(2.5) into the Python console, 6.25 is returned
        - *Doctest examples* – refers to the above examples
    - The function docstring is indented inside the function header, as a visual indicator that it is part of the overall function definition
- *Body* – the final line, return x ** 2
    - Code that is executed when the function is called
    - Also indented like the function docstring
    - Uses keyword return, which signals the *return statement*
        - Form of return statement: return <expression>
    - When a return statement is executed,
        1. The <expression> is evaluated, producing a value
        2. That value is then returned to wherever the function was called
            - No more code in the function body is executed after this point

What Happens When a Function is Called?

- Suppose we've defined square as above, and then call it in the Python console:
  >>> square(2.5)
- When we press Enter, the Python interpreter evaluates the function call by doing the following:
    1. Evaluate the argument 2.5, and then assign 2.5 to the function parameter x
    2. Evaluate the body of the square function, by doing:
        a. First evaluate x ** 2, which is 6.25 (since x refers to the value 2.5)
        b. Then stop executing the function body, and return the value 6.25 back to the Python console
    3. The function call square(2.5) evaluates to 6.25, and this is displayed on the screen
- When we call square *twice* in the same expression, i.e.
  >>> square(2.5) + square(-1.0)
    1. Python evaluates the operands to + in left-to-right order, so evaluate square(2.5) first
        a. Evaluate 2.5, and then assign 2.5 to the function parameter x
        b. Evaluate the body of the square function, by doing:
            i. First evaluate x ** 2, which is 6.25 (since x refers to 2.5)

ii. Then stop executing the function body, and return the value 6.25 back to the Python console

2. Nothing is displayed yet. There's still square(-1.0) to be evaluated
    a. Evaluate -1.0, and then assign -1.0 to the function parameter x
    b. Evaluate the body of the square function, by doing:
        i. First evaluate x ** 2, which is 1.0 (since x refers to -1.0)
        ii. Then stop executing the function body, and return the value 1.0 back to the Python console
3. Now the expression to evaluate has been simplified to 6.25 + 1.0, which evaluates to 7.25. This value is displayed on the screen

Defining Functions in Files

- We save functions in files so that we can reuse them across multiple sessions in the Python console (and in other files)

Defining Functions with Multiple Parameters

- Distance formula: $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
- When we write the function above in Python, the function will take 2 inputs:
    - Each input is a tuple of two floats, representing the *x*- and *y*-coordinates of each point
- Function header and docstring:
    - def calculate_distance(p1: tuple, p2: tuple) -> float:

```
"""Return the distance between points p1 and p2

p1 and p2 are tuples of the form (x, y), where the x- and y-coordinates
are points.

>>> calculate_distance((0, 0), (3.0, 4.0))
5.0
"""
```

- In order to use the above formula, we need to extract the coordinates from each point

```
x1 = p1[0]
y1 = p1[1]
x2 = p2[0]
y2 = p2[1]
```

- After having the four coordinates, we can apply the formula and return the result

```
return ((x1 – x2) ** 2 + (y1 – y2) ** 2) ** 0.5
```

Function Reuse

- For the above function body, we can reuse the square function defined above instead of ** 2
    - def calculate_distance(p1: tuple, p2: tuple) -> float:

```
"""Return the distance between points p1 and p2.

p1 and p2 are tuples of the form (x, y), where the x- and y-coordinates
are points.

>>> calculate_distance((0, 0), (3.0, 4.0))
5.0
"""
x1 = p1[0]
y1 = p1[1]
x2 = p2[0]
y2 = p2[1]
return (square(x1 – x2) + square(y1 – y2)) ** 0.5
```

- It is essential to organize our code into different functions as our programs grow larger

2.3 Local Variables and Function Scope

Intro

- A function call can only access its own variables, but not variables defined within other functions

Example 1: Introducing Local Variable Scope

- Example:

```
def square(x: float) -> float:
    """Return x squared.

    >>> square(3.0)
    9.0
    >>> square(2.5)
    6.25
    """
    return x ** 2
```

- The parameter x is a *variable* that is assigned a value based on when the function was called
    - This variable cannot be accessed from outside the body
- *Local variable* – variable limited to the function body
    - i.e. x
- *Scope* – places in the code where a variable can be accessed
    - A *local variable* of a function is a variable whose scope is the body of that function
- Example:

```
>>> n = 10.0
>>> result = square(n + 3.5)
```

    - 13.5 is assigned to the parameter x
    - Incorrect memory model diagram:

Variable	*Value*
n	10.0
x	13.5

    - We group the variables together based on whether they are introduced in the Python console or inside a function:

main (console)	
Variable	*Value*
n	10.0

square	
Variable	*Value*
x	13.5

    - We use the name _main_ to label the table for variables defined in the Python console
    - Inside the body of square, the *only* variable that can be used is x
        - At the point that the body of square is evaluated, only the "square" table in the memory model is active
    - Outside in the Python console, the *only* variable that can be used is n
        - After square returns and we're back to the Python console, the "square" table is no longer accessible, and only the _main_ table is active

Example 2: Duplicate Variable Names

- Suppose we modify our example above to use x instead of n in the Python console:

```
>>> x = 10.0
>>> result = square(x + 3.5)
```

    - This does *not* modify the x variable in the Python console
        - They are different variables though they shared the same name

main (console)

square

Variable	*Value*
x	10.0

Variable	*Value*
x	13.5

Example 3: (Not) Accessing Another Function's Variables

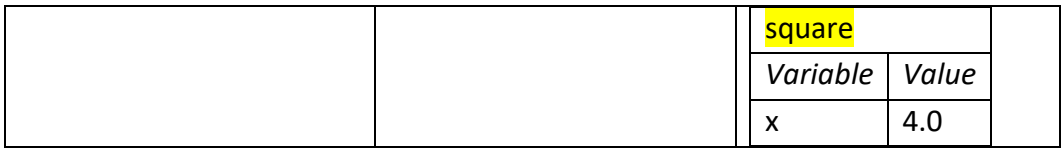
- Example consisting two functions:

```
def square(x: float) -> float:
    """Return x squared.

    >>> square(3.0)
    9.0
    >>> square(2.5)
    6.25
    """
    return x ** 2

def square_of_sum(numbers: list) -> float:
    """Return the square of the sum of the given numbers."""
    total = sum(numbers)
    return square(total)
```

  - Calling the function:

```
>>> nums = [1.5, 2.5]
>>> result = square_of_sum(nums)
>>> result
16.0
```

Right before square_of_sum is called (from console)	Right before square is called (from square_of_sum)	Right before square returns
main *Variable* | *Value* nums | [1.5, 2.5]	_main_ *Variable* | *Value* nums | [1.5, 2.5] square_of_sum *Variable* | *Value* numbers | [1.5, 2.5] total | 4.0	_main_ *Variable* | *Value* nums | [1.5, 2.5] square_of_sum *Variable* | *Value* numbers | [1.5, 2.5] total | 4.0



- The list [1.5, 2.5] is passed from the console to square_of_sum, and the number 4.0 is passed from square_of_sum to square
- Though the value 4.0 is passed from total to x, it would not work calling total in square
- Even if square of sum is still active, total cannot be accessed from square

- Python's rule for local scope: a local variable can only be accessed in the function body it is defined
  - Prevents us from accidentally using a variable from a completely different function when working on a function

2.4 Importing Modules

Intro

- *Modules* – Python code files
- The modules are not automatically loaded

The Import Statement

- *Import statement* – what we use to load a Python module
  - Syntax: import <module_name>
- i.e. to load the math module, we type

```
>>> import math
```

- To access the function, we use dot notation

```
>>> math.log2(1024)
10.0
```

- Call the built-in function dir on the module to see a list of functions and other variables defined in the module
  - i.e. dir(math)

The Datetime Module

- Provides not just functions but new *data types* for representing time-based data
- *date* – a data type that represents a specific date
  - i.e.

```
>>> import datetime
>>> canada_day = datetime.date(1867, 7, 1)  # Create a new date
>>> type(canada_day)
```

```
<clad 'datetime.date'>
>>> term_start = datetime.date(2020, 9, 10)
>>> datetime.date.weekday(term_start)  # Return the day of the week of the date
3  # 0 = Monday, 1 = Tuesday, etc.
```

- Note the double use of dot notation
  - datetime.date is the date type being accessed, and .weekday accesses a method of the data type

- We can compare dates for equality using == and chronological order (e.g. < for comparing one date comes before another)
- We can also subtract dates
  - i.e.

```
>>> term_start - canada_day
datetime.timedelta(days=55954)
```

  - The difference between two dates is an instance of the datetime.timedelta data type, which is used to represent an interval of time

2.5 The Function Design Recipe

The Function Design Recipe by Example

1. Writing example uses
   a. A good name for a function is a short answer to the question "What does the function do?"
   b. Write 1 or 2 examples of calls to the function and the expected return values
      i. Include an example of a standard case (as opposed to a tricky case)
   c. Put the examples inside a triple-quoted string that is indented
   d. i.e.

```
"""
>>> is_even(2)
True
>>> is_even(17)
False
"""
```

2. Write the function header
   a. Above the docstring
   b. Choose a meaningful name for each parameter
   c. Include the type contract (the types of the parameters and return value)
   d. i.e.

```
def is_even(value: int) -> bool:
    """
    >>> is_even(2)
    True
    >>> is_even(17)
    False
    """
```

3. Write the function description
    a. Add a description of what the function does before the examples
    b. Make sure the purpose of each parameter is clear
    c. Describe the return value
    d. i.e.

```
def is_even(value: int) -> bool:
    """Return whether value is even.

    >>> is_even(2)
    True
    >>> is_even(17)
    False
    """
```

4. Implement the function body
    a. Indent it to match the docstring
    b. Review the examples and consider how we determined the return values
    c. i.e.

```
def is_even(value: int) -> bool:
    """Return whether value is even.

    >>> is_even(2)
    True
    >>> is_even(17)
    False
    """
    return value % 2 == 0
```

5. Test the function
    a. Test all examples
        i. Try with some tricky cases as well
    b. Test by calling it in the Python console

c. If we encounter any errors/incorrect return values, make sure that our tests are correct
d. Go back to Step 4 and try to identify and fix any possible errors in the code
    i. Called *debugging*

The Importance of Documenting our Functions

- The Function Design Recipe places a large emphasis on developing a precise and detailed function header and docstring before writing any code for the function body
- Forces us to write out the function header, docstring, and examples reinforces our understanding about what we need to do
- The function header and docstring serve as *documentation* for the function
    - Communicating to others and our future self what the function is supposed to do

2.6 Testing Functions I: Doctest and Pytest

Doctests: Basic Examples in Docstrings

- *Manual testing* – requires human interaction to complete
    - i.e. importing the function into the Python console, and then manually evaluate each doctest example one at a time and compare the output with the expected output in the docstring
- Use the Python library doctest, which can automatically extract doctest examples from docstrings and convert them into runnable tests
- To use doctest, add the following code to the bottom of any Python file

```
if __name__ == '__main__':
        import doctest  # import the doctest library
        doctest.testmod()  # run the tests
```

    - We will receive a report about which tests failed
- In order to use doctest, our docstring examples must be correctly formatted and valid Python code

Creating Test Suites with Pytest

- We use pytest to write our tests in a separate file, thus including an exhaustive set of tests without cluttering our code files
- i.e. suppose we have defined the following function in a files trues.py

```
# In files trues.py

def has_more_trues(booleans: list) -> bool:
```

```
    """Return whether booleans contain more True

    >>> has_more_trues([True, False, True])
    True
    >>> has_more_trues([True, False, False])
    False
    """
    #  Function body omitted
```

- Write tests in a new file and name it test_trues.py
- *Unit test* – a block of code that checks for the correct behaviour of a function for one specific input
- *Test suite* – a collection of tests that check the behaviour of a function or (usually small) set of functions
- Every test file contains a test suite
- In Python, we express a unit test as a function whose name starts with the prefix:
  test_
- The body of the function contains an assert statement, which is used to check whether some boolean expression is True or False
- i.e.

```
# In file test_trues.py

from trues import has_more_trues

def test_mixture_one_more_true() -> None:
    """Test has_more_trues on a list with a mixture of True and False, with one
    more True than False.
    """
    assert has_more_trues([True, False, True])

def test_mixture_one_more_false() -> None:
    """Test has_more_trues on a list with a mixture of True and False, with one
    more False than True.
    """
    assert not has_more_trues([True, False, False])
```

- The unit test functions are similar to functions we've defined previously, with a few differences:

    - Each test name and docstring documents what the test is by describing the test input
    - The return type of the test function is None, which is a special type that indicates that no value at all is returned by the function.
        - There is no return statement – instead, there's an assert
- An assert statement with the form assert <expression> does the following when executed:
    1. First, it evaluates <expression>, which should produce a boolean value
    2. If the value is True, nothing else happens, and the program continues onto the next statement
    3. If the value is False, an AssertionError is raised. This sygnals to Pytest that the test has failed.
- When pytest "runs" a unit test, it calls a test function like test_mixture_one_more_true
    - If the function call ends without raising an AssertionError, the test *passes*
    - If the function call does raise an AssertionError, the test *fails*
- A single unit test function can contain multiple assert statements
- To run our unit test functions with pytest, we need to first import pytest and then call a specific test function

```
# At the bottom of test_trues.py

if __name__ == '__main__':
        import pytest
        pytest.main(['test_trues.py'])
```

2.7 Type Conversion Functions

Intro

- We can convert values between different data types
- i.e.

```
>>> int('10')
10
>>> float('10')
10.0
>>> bool(1000)
True
>>> bool(0)
False
```

```
>>> list({1, 2, 3})
[1, 2, 3]
>>> set([1, 2, 3])
{1, 2, 3}
>>> dict([('a', 1), ('b', 2), ('c', 3')])
{'a': 1, 'b': 2, 'c': 3}
```

- Every value of the data types we've studied so far has a string representation
  - i.e.

```
>>> str(10)
'10'
>>> str(True)
'True'
>>> str([1, 2, 3])
'[1, 2, 3]'
```

Warning: Conversion Errors

- Not all values of one type can be converted into another
  - i.e.

```
>>> int('David')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'David'
>>> list(1000)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Preview: Creating Values of Arbitrary Data Types

- range
  - We could call range to create a sequence of numbers
  - When calling range by itself, this happens

```
>>> range(5, 10)
range(5, 10)  # rather than [5, 6, 7, 8, 9]
```

  - range is a type conversion function
    - Python has a range data type that is distinct from lists
  - i.e.

```
>>> five_to_nine = range(5, 10)
>>> type(five_to_nine)
```

```
<class 'range'>
>>> five_to_nine == [5, 6, 7, 8, 9]
False
```

- datetime.date
    - Example:

```
>>> import datetime
>>> canada_day = datetime.date(1867, 7, 1)  # Create a new date
>>> type(canada_day)
<clad 'datetime.date'>
```

    - The data type datetime.date is called on 3 arguments instead of 1
    - datetime.date is called to *create* a new date value given three arguments (the year, month, and day)
- We'll be able to take *any* data type – even ones we defined ourselves – and create values of that type by calling the data type as a function

2.8 Application: Representing Text

Intro

- Numbers represent textual data via functions
- *Bit* – what we call each 0 or 1
- *ASCII* – a function with domain {0, 1, …, 127} whose codomain is the set of all possible characters
    - Length-7 sequences of bits
        - Can represent $2^7 = 128$ different characters
    - The function is *one-to-one*, meaning no two numbers map to the same character
    - This standard covered all English letters (lowercase and uppercase), digits, punctuation, and various others (e.g. to communicate a new line)
        - i.e. 65 is mapped to 'A' and 126 is mapped to '~', etcl
- Computer scientists extended ASCII from length-7 to length-8 sequences of bits, and hence its domain increased to size 256 ({0, 1, …, 255}
    - Allowed "extended ASCII" to support some other characters used in similar Latin-based languages
        - i.e. 'é' (233), 'ō' (246), '©' (169), etc.
- The latest standard, Unicode, uses *up to 32 bits* that gives us a domain of {0, 1, …, 2^{32}-1}, over 4 billion different numbers
    - This number is larger than the number of distinct characters in use across all different languages
    - There are several *unused numbers* in the domain of Unicode

    - The unused numbers are being used to *map to emojis*
        - An emoji may appear as a different emoji on another device
    - The process involves *submitting a proposal for a new emoji* and computer scientists supporting newly approved emojis by updating their software

Python's Unicode Conversion Functions

- Python has 2 built-in functions that implement the (partial) mapping between characters and their Unicode number
- *ord* – takes a single-character string and returns its Unicode number as an int
    - i.e.

```
>>> ord('A')
65
>>> ord('é')
233
>>> ord('猿')
29503
```

- *chr* – computes the *inverse* of ord
    - Given an integer representing a Unicode number, chr returns a string containing the corresponding character
    - i.e.

```
>>> chr(65)
'A'
>>> chr(233)
'é'
>>> chr(29503)
'猿'
```

- String ordering comparisons (<, >) are based on Unicode numeric values
    - i.e. the Unicode value of 'Z' is 90 and the Unicode value of 'a' is 97

```
>>> 'Z' < 'a'
True
>>> 'Zebra' < 'animal'
True
```

- Sorting a collection of strings can seem alphabetical, but treats lowercase and uppercase letters differently
    - i.e.

```
>>> sorted({'David', 'Mario', 'Jacqueline'})
['David', 'Jacqueline', 'Mario']
>>> sorted({'david', 'Mario', 'Jacqueline'})
```

['Jacqueline', 'Mario', 'david']

3.1 Propositional Logic

Propositions

- *Propositional logic* – an elementary system of logic that is a crucial building block underlying other
- *Proposition* – a statement that is either True or False
    - i.e. 2 + 4 = 6; 3 – 5 > 0; Python's implementation of list.sort is correct on every input list; etc.
- *Propositional variables* – variables that represent propositions
    - By conventions, propositional variable names are lowercase letters starting at p
- *Propositional/Logical operator* – an operator whose arguments must all be either True or False
- *Propositional formula* – an expression that is built up from propositional variables by applying the propositional operators

The Basic Operators **NOT**, **AND**, **OR**

- *NOT* – unary operator, also called "negation", denoted by the symbol ¬
    - Negates the truth value of its input
    - i.e. if p is True, then ¬p is False, and vice versa
- *AND* – binary operator, also called "conjunction", denoted by the symbol ∧
    - Returns True when both its arguments are True
- *OR* – binary operator, also called "disjunction", denoted by the symbol ∨
    - Returns True if one or both of its arguments are True
    - The *inclusive or*

The Implication Operator

- *Implication* – represented by the symbol ⇒
    - The implication p ⇒ q asserts that whenever p is True, q must also be True
        - *Hypothesis* – p
        - *Conclusion* – q
    - *Vacuous truth* – p is False but p ⇒ q is True
    - Formula p ⇒ q has 2 *equivalent* formulas
        - The formula is only False when p is True and q is False
        - *Equivalent* – the two formulas have the same truth values
            - i.e. the formulas will either both be True or both be False
        - $\neg p \vee q$

- If p is False then p ⇒ q is True
- If p is True then q must be True as well
- Only False when p is True and q is False

▪ $\neg q \Rightarrow \neg p$

- If q doesn't occur, then p cannot have occurred either
- Only False when p is True and q is False
- *Contrapositive* for p ⇒ q

o *Converse* – the implication q ⇒ p

▪ Obtained by switching the hypothesis and conclusion
▪ *Not* logically equivalent to the original implication

- In Python, there is no operator or keyword that represents implication directly
  - To express an implication in Python, use the first equivalent form from above
    - Write p ⇒ q as $\neg p \vee q$

Biconditional ("If and Only If")

- *Biconditional* – denoted by $p \Leftrightarrow q$
  - Returns True when p ⇒ q and its converse q ⇒ p are both True
  - Abbreviation for p ⇒ q ⋀ q ⇒ p
  - "if p then q, and if q the p"
  - "p if and only if q"
  - "p iff q"
- In Python, we use == to determine whether two boolean values are the same

Summary

Operator	Notation	English	Python Operation
NOT	$\neg p$	p is not true	not p
AND	$p \wedge q$	p and q	p and q
OR	$p \vee q$	p or q (or both)	p or q
Implication	$p \Rightarrow q$	if p, then q	not p or q
Biconditional	$p \Leftrightarrow q$	p if and only if q	p == q

3.2 Predicate Logic

Intro

- *Predicate* – a statement whose truth value depends on 1 or more variables from any set
  - Codomain of the function: {True, False}
  - Use uppercase letters starting from P to represent predicates
  - i.e. P(x) is defined to be the statement "x is a power of 2"

        - P(8) is True and P(7) is False
- Predicates can depend on more than 1 variable
    - i.e. Q(x, y) means "$x^2 = y$"
        - Q(5, 25) is True since $5^2 = 25$, but Q(5, 24) is False
- We must always give the domain of a predicate as part of its definition
    - i.e. P(x): "x is a power of 2,", where $x \in \mathbb{N}$

Quantification of Variables

- A predicate by itself does not have a truth value
    - i.e. "x is a power of 2" is neither True nor False, since we don't know that value of x
    - Setting x = 8 in the statement, the statement is now True
- Most of the times, we care about some aggregation of the predicate's truth values over *all* elements of its domain
    - i.e. "Every real number x satisfies the inequality $x^2 - 2x + 1 \geq 0$" makes a claim about *all possible* values of x
- *Quantifier* – modifies a predicate by specifying how a certain variable should be interpreted
- Existential quantifier
    - *Existential quantifier* – written as ∃
        - "*There exists* an element in the domain that satisfies the given predicate"
        - i.e. $\exists x \in \mathbb{N}, x \geq 0$

          "There exists a natural number x that is greater than or equal to 0."
            - True because when x = 1, x ≥ 0
    - There has to be *at least 1* element of the domain satisfying the predicate
        - Doesn't say exactly how many elements do
    - $\exists x \in S$ – a big OR that runs through all possible values for x from the domain S
        - i.e. for the previous example,

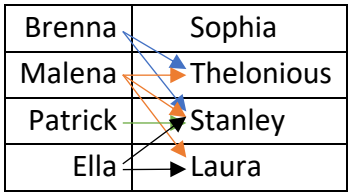
$$(0 \geq 0) \vee (1 \geq 0) \vee (2 \geq 0) \vee \ldots$$

- Universal quantifier
    - *Universal quantifier* – written as ∀
        - "*Every* element in the domain satisfies the given predicate"
        - i.e. $\forall x \in \mathbb{N}, x \geq 0$

          "Every natural number x is greater than or equal to 0"
            - True because the smallest natural number is 0 itself
        - i.e. $\forall x \in \mathbb{N}, x \geq 10$ is False
    - $\forall x \in S$ – a big AND that runs through all possible values of x from S
        - i.e. for the first example,

$$(0 \geq 0) \wedge (1 \geq 0) \wedge (2 \geq 0) \wedge \ldots$$

- Example: We define Loves(a, b) is a binary predicate that is True whenever person a loves person b
  - A = {Ella, Patrick, Malena, Breanna}
  - B = {Laura, Stanley, Thelonious, Sophia}
  - A line between 2 people indicates that the person on the left loves the person on the right



  - Consider the following statements:
    - $\exists a \in A, Loves(a, \text{Thelonious})$ means "there exists someone in A who loves Thelonious"
      - True (Malena loves Thelonious)
    - $\exists a \in A, Loves(a, \text{Sophia})$ means "there exists someone in A who loves Sophia"
      - False (no one loves Sophia)
    - $\forall a \in A, Loves(a, \text{Stanley})$ means "every person in A loves Stanley"
      - True (all 4 people in A loves Stanley)
    - $\forall a \in A, Loves(a, \text{Thelonious})$ means "every person in A loves Thelonious"
      - False (Ella does not love Thelonious)

Python Built-ins: Any and All

- *Any* – built-in function that represent logical statements using the existential quantifier
  - Takes a collection of boolean values and returns True when there exists a True value in the collection
  - i.e.

```
>>> any([False, False, True])
True
>>> any([])  # An empty collection has no True values
False
```

  - i.e. suppose we're given a set of strings S and wish to determine whether any of them start with the letter 'D',

```
>>> strings = ['Hello', 'Goodbye', 'David']
>>> any([s[0] == 'D' for s in strings])
True
```

    - Parallels between mathematical statements and Python expressions:
        - ∃ corresponds to calling the any function
        - $s \in S$ corresponds to for s in strings
        - s[0] = 'D' corresponds to s[0] == 'D'
- *All* – built-in function that can be used as a universal quantifier
    - Takes a collection values and evaluates to True when every element has the value True
    - i.e. to express $\forall s \in S$, s[0] = 'D' in Python,

```
>>> strings = ['Hello', 'Goodbye', 'David']
>>> all([s[0] == 'D' for s in strings])
False
```

- Since Python is limited on the size of collections, we cannot easily express existential statement quantified over infinite domains like $\mathbb{N}$ or $\mathbb{R}$

Writing Sentences in Predicate Logic

- A general formula in predicate logic is built up using the existential and universal quantifiers, the propositional operators, and arbitrary predicates
- *Sentence* – a formula with no unquantified variables
    - i.e.
      the formula $\forall x \in \mathbb{N}, x^2 > y$ is not a sentence as y is not quantified
      the formula $\forall x, y \in \mathbb{N}, x^2 > y$ is a sentence
- Avoid using commas
    - "Does the comma mean 'and' or 'then'?"
        - i.e. P(x), Q(x)
    - *Never use the comma to connect propositions*
        - Use ∧ and ⟹
    - English is too ambiguous for fields of computer science
        - i.e. "or" can be inclusive or exclusive
    - Where to use comma:
        - Immediately after a variable quantification, or separating two variables with the same quantification
        - Separating arguments to a predicate
        - i.e. $\forall x, y \in \mathbb{N}, \forall z \in \mathbb{R}, P(x, y) \implies Q(x, y, z)$

Manipulating Negation

- Given any formula, we can state its negation by preceding it by a ¬ symbol
    - i.e. $\neg(\forall x, y \in \mathbb{N}, \forall z \in \mathbb{R}, P(x, y) \implies Q(x, y, z))$
    - Hard to understand if we try to transliterate each part separately

- Given a formula using negations, we apply some *simplification rules* to "push" the negation symbol closer to the individual predicates:
    - $\neg(\neg p)$ becomes p
    - $\neg(p \vee q)$ becomes $(\neg p) \vee (\neg q)$
    - $\neg(p \wedge q)$ becomes $(\neg p) \wedge (\neg q)$
    - $\neg(p \implies q)$ becomes $p \wedge (\neg q)$
    - $\neg(p \iff q)$ becomes $(p \wedge (\neg q)) \vee ((\neg p) \wedge q))$
    - $\neg\big(\exists x \in S, P(x)\big)$ becomes $\forall x \in S, \neg P(x)$
    - $\neg\big(\forall x \in S, P(x)\big)$ becomes $\exists x \in S, \neg P(x)$

3.3 Filtering Collections

Expressing Conditions in Predicate Logic

- Consider the statement: "Every natural number n greater than 3 satisfies the inequality $n^2 + n \geq 20$."
    - "Greater than 3" is a *condition* that modifies the statement, limiting the original domain of n (the natural numbers) to a smaller subset (the natural numbers greater then 3)
    - Two ways to represent such conditions in predicate logic:
        - Define a new set
            - i.e. $S_1 = \{n \mid n \in \mathbb{N}\ and\ n > 3\}, \forall n \in S_1, n^2 + n \geq 10$
        - Use an implication to express the condition
            - Rewrite the original statement using an "if … then …" structure: "For every natural number n, if n is greater than 3 then n satisfies the inequality $n^2 + n \geq 20$.
            - Translating the above into predicate logic:
            $$\forall n \in \mathbb{N}, n > 3 \implies n^2 + n \geq 20$$
            - $n > 3 \implies$ has a filtering effect, due to the *vacuous truth* case of implication
                - i.e. For the values n ∈ {0, 1, 2}, the hypothesis of the implication is False, and so for these values the implication itself is True
            - Since the overall statement is universally quantified, these vacuous truth cases don't affect the truth value of the statement
- The "forall-implies" structure arises naturally when a statement is universally quantified

Filtering Collections in Python

- *Filter operation* – an operation that takes a collection of data and returns a new collection consisting of the elements in the original collection that satisfy some predicate
    - {<expression> for <variable> in <collection> if <condition>}
    - This form of set comprehension behave the same way as the ones we studied last chapter, except that <expression> only gets evaluated for the values of the variable that make the condition evaluate to True
    - i.e.

```
>>> numbers = {1, 2, 3, 4, 5}  # Initial collection
>>> {n for n in numbers if n > 3}  # Pure filtering: only keep elements > 3
{4, 5}
>>> {n * n for n in numbers if n > 3} # Filtering with a data transformation
{16, 25}
```

- By combining filtering comprehensions with aggregation functions, we can limit the scope of an aggregation
    - i.e.

```
>>> numbers = {1, 2, 3, 4, 5}
>>> sum({n for n in numbers if n % 2 == 0})  # Sum of only the even
numbers
6
```

- The keyword if used in this syntax is connected to our use of implication
    - i.e. $\forall n \in N, n > 3 \Longrightarrow n\text{^}2 + n \geq 20$ in Python:

```
>>> numbers = {1, 2, 3, 4, 5, 6, 7, 8}
>>> all({n ** 2 + n >= 20 for n in numbers if n > 3})
True
```

3.4 Conditional Execution

The *if* Statement

- *If statement – compound statement* that expresses conditional execution of code
    - *Compound statement* – contains other statements within it
    - Syntax:

```
if <condition>:
        <statement>
        …
else:
        <statement>
        …
```

- *If condition* – the <condition> following if that evaluates to a boolean
  - Analogous to the hypothesis of an implication
- *If branch* – statements under the if
- *Else branch* – statements under the else
- When an if statement is executed,
  - The if condition is evaluated, producing a boolean value
  - If the condition evaluates to True, then the statements in the if branch are executed.
  - If the condition evaluates to False, then the statements in the else branch are executed instead
- i.e.

```
def get_status(scheduled: int, estimated: int) -> str:
    """Return the flight status for the given scheduled and estimated
    departure times

    The times are given as integers between 0 and 23 inclusive,
    representing the hour of the day.

    The status is either 'On time' or 'Delayed'.

    >>> get_status(10, 10)
    'On time'
    >>> get_status(10, 12)
    'Delayed'
    """
```

- If we only need to calculate a bool for whether the flight is delayed, simply return estimated <= scheduled
- We use if statements to execute different code based on these cases:
  - i.e.

```
def get_status(scheduled: int, estimated: int) -> str:
    """..."""
    if estimated <= scheduled:
        return 'On time'
    else:
        return 'Delayed'
```

  - This uses the boolean expression we identified earlier to trigger different return statements to return the correct string

Code with More Than Two Cases

- i.e. whenever a flight is delayed by more than 4 hours, the airline cancels the flight

```
def get_status_v2(scheduled: int, estimated: int) -> str:
    """Return the flight status for the given scheduled and estimated
    departure times

    The times are given as integers between 0 and 23 inclusive, representing
    the hour of the day.

    The status is either 'On time', 'Delayed', or 'Cancelled'

    >>> get_status_v2(10, 10)
    'On time'
    >>> get_status_v2(10, 12)
    'Delayed'
    >>> get_status_v2(10, 15)
    'Cancelled"
    """
```

- We can express subcases using nested if statements
  - i.e.

```
def get_status_v2(scheduled: int, estimated: int) -> str:
    """..."""
    if estimated <= scheduled:
        return 'On time'
    else:
        if estimated – scheduled <= 4:
            return 'Delayed'
        else:
            return 'Cancelled'
```

- Excessive nesting of statements can make code difficult to read and understand
- *elif* – short for "else if"
  - Syntax:

```
if <condition1>:
    <statement>
    …
elif <condition2>:
    <statement>
    …
```

```
    … # [any number of elif conditions and branches]
    else:
        <statement>
        …
```

- When executed,
  - The if condition (<condition1>) is evaluated, producing a boolean value
  - If the condition evaluates to True, then the statements in the if branch are executed. If the condition evaluates to False, the next elif condition is evaluated, producing a boolean
  - If that condition evaluates to True, then the statements in that elif branch are executed. If that condition evaluates to False, then the next elif condition is evaluated. This step repeats until either one of the elif conditions evaluate to True, or all of the elif conditions have evaluated to False
  - If neither the if condition nor any of the elif conditions evaluate to True, then the else branch executes
- i.e.

```
def get_status_v3(scheduled: int, estimated: int) -> str:
    """…"""
    if estimated <= scheduled:
        return 'On time'
    elif estimated – scheduled <= 4:
        return 'Delayed'
    else:
        return 'Cancelled'
```

  - Equivalent to the nested version but easier to read

Testing All the Branches

- We need to design our unit tests so that each possible execution path is used at least once
- *White box testing* – testing where we "see through the box" and therefore can design tests based on the source code itself
- *Black box testing* – tests created without any knowledge of the source code
  - No knowledge of the different paths the code can take
- In our doctests for get_status_v3, we chose three different examples, each corresponding to a different possible case of the if statement
- *Code coverage* – the percentage of lines of program code that are executed when a set of tests are used on that program

- Metric used to access the quality of tests

Building on Our Example

- i.e. write a function that determines how many flights are cancelled in a day. We are provided with a dictionary in the form {flight_number: [scheduled, estimated]}

```
>>> flights = {'AC110': [10, 12], 'AC321": [12, 19], 'AC999': [1, 1]}
>>> flights['AC110']
[10, 12]
```

  - We can call our get_status_v3 function for flights
    - i.e.

```
>>> flight_ac110 = flights['AC110']
>>> get_status_v3(flight_ac110[0], flight_ac110[1])
'Delayed'
```

  - Instead of specifying the flight number ourselves, we could substitute in different flight numbers based on our data using comprehensions
    - i.e.

```
>>> {k for k in flights}
{'AC299', 'AC110', AC321'}
>>> {get_status_v3(flights[k][0], flights[k][1] == 'Cancelled' for k in flights)}
{False, True}
>>> [get_status_v3(flights[k][0], flights[k][1] == 'Cancelled' for k in flights)]
[False, True, False]
```

    - The first set comprehension does not tell us if the flight was cancelled
    - For the second set comprehension we could see that there was at least 1 flight cancelled
    - The list comprehension tells us that exactly 1 out of 3 flights were cancelled
  - We could combine the first set comprehension with the second
    - i.e.

```
>>> {k for k in flights if get_status_v3(flights[k][0], flights[k][1] == 'Cancelled')}
{'AC321'}
>>> [k for k in flights if get_status_v3(flights[k][0], flights[k][1] == 'Cancelled')]
['AC321']
```

  - To convert the set into an integer, use the built-in len function on the set

- i.e.

```
def count_cancelled(flights: dict) -> int:
    """Return the number of cancelled flights for the given
    flight data.

    flights is a dictionary where each key is a flight ID, and
    whose corresponding value is a list of 2 numbers, where
    the first is the scheduled departure time and the second is
    the estimated departure time.

    >>> count_cancelled({'AC110': [10, 12], 'AC321": [12, 19],
    'AC999': [1, 1]})
    1
    """
    cancelled_flights = {k for k in flights if
    get_status_v3(flights[k][0], flights[k][1] == 'Cancelled')}
    return len(cancelled_flights)
```

3.5 Simplifying If Statements

<u>Computing Booleans: When *if* Statements Aren't Necessary</u>

- In cases where we need a boolean value, *it is often simpler to write an expression to calculate the value directly, rather than using if statements*
- i.e.

```
def is_even(n: int) -> bool:
    """Return whether n is even (divisible by 2)."""
    if n % 2 == 0:
        return True
    else:
        return False
```

can be simplified to

```
def is_even(n: int) -> bool:
    """..."""
    return n % 2 == 0
```

- i.e. a more complex example

```
def mystery(x: lst, y: lst) -> bool:
    if x == []:
        if y == []:
```

```
                return True
            else:
                return False
        else:
            if y = []:
                return False
            else:
                return True
```

simplify the nested if statements:

```
def mystery(x: lst, y: lst) -> bool:
    if x == []:
        return y == []
    else:
        return y != []
```

combining the cases,

```
def mystery(x: lst, y: lst) -> bool:
    return (x == [] and y == []) or (x != [] and y != [])
```

for readability,

```
def mystery(x: lst, y: lst) -> bool:
    both_empty = x == [] and y == []
    both_non_empty = x != [] and y != []
    return both_empty or both_non_empty
```

Using *if* Statements

- Prefer using a sequence of elifs rather than nested if statements
- Write conditions from most specific to most general
    - Order matters for conditions, since they are checked one at a time in top-down order

3.6 if __name__ == '__main__'

Import Statements

- Allows the program that executes the import statement to access the functions and data types defined within that module
- By default, *all* statements in the imported module are executed, not just function and data type definitions
- Without the if statement, all the doctests inside the imported modules will be run, even though they are not relevant for a program that just wants to use the imported modules

Enter __name__

- *__name__* - a special variable for each module when a program is run
    - Double underscore denotes special variable or function names
    - i.e. the __name__ attribute of math is 'math'

```
>>> import math
>>> math.__name__
'math'
```

- When we *run* a module, the Python interpreter overrides the default module __name__ and instead sets it to the special string '__main__'
- Checking the __name__ variable is a way to determine if the current module is being run, or whether it's being imported by another module
- if __name__ == '__main__' – "Execute the following code if this module is being run, and ignore the following code if this module is being imported by another module"

Organizing a Python File

- *Main block* – all the code under if __name__ == '__main__'
- The only code that goes outside of the main block are:
    - Import statements (import …)
    - Function definitions (def …)
    - Data type definitions (class …)
- Other code, like code for running doctest or pytest, goes inside the main block so that it is only executed when the module is run, and not when it is imported
- The main block goes at the bottom of the module

3.7 Function Specifications

Function Specifications and Correctness

- A *specification* for a function consists of 2 parts:
    - A description of what values the function takes as valid inputs
        - We can represent this description as a set of predicates, where a valid input must satisfy all these predicates
        - *Preconditions* – how we call these predicates
    - A description of what the function returns/does, in terms of its inputs
        - We can represent this description as a set of predicates that must all be satisfied by the return value of the function
        - *Postconditions* – how we call these predicates
- A function implementation is *correct* when the following holds:

    - *For all inputs that satisfy the specification's preconditions, the function implementation's return value satisfies the specification's postconditions*
- The person implementing the function needs to make sure that the code correctly returns or does what the specification says
    - They do not need to worry about exactly how the function will called, and can *assume* that the function's input is always valid
- The person calling the function needs to make sure that they call the function on valid inputs
    - They do not need to worry about exactly how the function is implemented, and can *assume* that the function works correctly

Simple Specifications

- Example:

```
def is_even(n: int) -> bool:
    """Return whether n is even.

    >>> is_even(1)
    False
    >>> is_even(2)
    True
    """
    # Body omitted
```

- The function's *type contract* and *description* forms a complete specification of this function's behaviour:
    - The type annotation of the parameter n tells us that the valid inputs to is_even are int values
        - This type annotation int specifies a *precondition* of the function
    - The type annotation for the return value tells us that the function will always return a bool
        - The function description Return whether n is even completes the specification by indicating how the return value is based on the input
        - These specify *postconditions* of the function
- is_even is implemented correctly when *for all* ints n, is_even(n) *returns a* bool *that is* True *when* n *is even, and* False *when* n *is not even*
- If this happens:

```
>>> is_even(4)
False
```

    - It's the implementer's fault

- If this happens and an error occurs

  >>> is_even([1, 2, 3])

  - It's the caller's fault

Preconditions in General

- Consider this function:

```
def max_length(strings: set) -> int:
    """Return the maximum length of a string in the set of strings.

    >>> max_length({'Hello',  'Mario, 'David Liu'})
    9
    """
    return max({len(s) for s in strings})
```

- When the set is empty, there would be an error
  - The implementer is at fault
    - The only description of "valid inputs" given is the type annotation set
      - An empty set is a set
  - We can add additional precondition to the function docstring:

```
def max_length(strings: set) -> int:
    """Return the maximum length of a string in the set of strings.

    Preconditions:
        ■ len(strings) > 0
    """
    return max({len(s) for s in strings})
```

    - Now, when we call max_length(empty_set), and receive an error, it is our fault for violating the precondition
- Checking preconditions automatically with python_ta
  - Preconditions can be turned into executable Python code
    - Use an assert statement as follows:

```
def max_length(strings: set) -> int:
    """Return the maximum length of a string in the set of
    strings.

    Preconditions:
        ■ len(strings) > 0
    """
```

```
    assert len(strings) > 0, 'Precondition violated: max_length
    called on an empty set'
    return max({len(s) for s in strings})
```

- Now, precondition is checked every time the function is called, with a meaningful error message when the precondition is violated

- We can also use the python_ta library to check preconditions

```
if __name__ == '__main__':
    import python_ta.contracts
    python_ta.contracts.DEBUG_CONTRACTS = False  # Disable
    contract debug messages
    python_ta.contracts.check_all_contracts()

    max_length(set())
```

  - The function we import, check_all_contracts, takes the function type contract and any preconditions it finds in the function docstring, and causes the function to check the preconditions every time the function is called

- check_all_contracts also checks the return type of each function

Preconditions as Assumptions and Restrictions

- Preconditions place restrictions on the user of the function
  - The onus is on them to respect these preconditions
- Besides ruling out the invalid input with a precondition, we can explicitly define some alternate function behaviour for this input:

```
def max_length(strings: set) -> int:
    """Return the maximum length of a string in the set of strings.

    Return 0 if strings is empty
    """
    if strings == set():
        return 0
    else:
        return max({len(s) for s in strings})
```

- Result:

```
>>> empty_set = set()
>>> max_length(empty_set)
0
```

3.8 The *typing* Module and Richer Type Annotations

Intro

- Recall the function max_length

```
def max_length(strings: set) -> int:
    """Return the maximum length of a string in the set of strings.

    Preconditions:
        ■ len(strings) > 0
    """
    return max({len(s) for s in strings})
```

  - >>> max_length ({1, 2, 3}) outputs an error despite the fact that our inputs are valid

Typing's Collection Types

- There are 4 collection types that we can import from the typing module

Type	Description
Set[T]	A set whose elements all have type T
List[T]	A list whose elements all have type T
Tuple[T1, T2, …]	A tuple whose first element has type T1, second element has type T2, etc.
Dict[T1, T2]	A dictionary whose keys are of type T1 and whose values are of type T2

- i.e.
  - {'hi', bye'} has type Set[str]
  - [1, 2, 3] has type List[int]
  - ('hello', True, 3.4) has type Tuple[str, bool, float]
  - {'a': 1, 'b': 2, 'c': 3} has type Dict[str, int]
- Improving max_length with the typing module

```
from typing import Set

def max_length(strings: Set[str]) -> int:
    """Return the maximum length of a string in the set of strings.

    Preconditions:
        ■ len(strings) > 0
```

```
    """
    return max({len(s) for s in strings})
```

- General collections
    - The above typing module's collection types are not needed when:
        - We don't care what's in the list
        - We want a list with elements of different types

3.9 Working with Definitions

Intro

- *Definitions* – what we use to express a long idea using a single term
- Let $n, d \in \mathbb{Z}$. We say that d *divides* n, or n *is divisible by* d, when there exists a $k \in \mathbb{Z}$ such that $n = dk$
    - In this case, we use the notation $d \mid n$ to represent "d divides n"
    - $\mid$ is a binary predicate
        - i.e. 3 | 6 is True; 4 | 10 is false
    - This definition permits $d = 0$
        - When $d = 0, d \mid n$ if and only if $n = 0$
- Expressing the statement "For every integer x, if x divides 10, then it also divides 100"
    - With the predicate:
        - $\forall x \in \mathbb{Z}, x \mid 10 \implies x \mid 100$
    - Without the predicate
        - Replace every instance of $d \mid n$ with $\exists k \in \mathbb{Z}, n = dk$
        - $\forall x \in \mathbb{Z}, (\exists k \in \mathbb{Z}, 10 = kx) \implies (\exists k \in \mathbb{Z}, 100 = kx)$
        - Each subformula in the parentheses has its own k variable, whose scope is limited by the parentheses
        - To emphasize their distinctness,
            - $\forall x \in \mathbb{Z}, (\exists k_1 \in \mathbb{Z}, 10 = k_1 x) \implies (\exists k_2 \in \mathbb{Z}, 100 = k_2 x)$
- Let $p \in \mathbb{Z}$. We say p is *prime* when it is greater than 1 and the only natural numbers that divide it are 1 and itself
- Define a predicate $IsPrime(p)$ to express the statement that "p is a prime number"
    - First part: "greater than 1"
    - Second part: "*if a number d divides p, then* $d = 1$ *or* $d = p$"
    - $IsPrime(p) : p > 1 \wedge (\forall d \in \mathbb{N}, d \mid p \Rightarrow d = 1 \vee d = p)$, where $p \in \mathbb{Z}$
    - To express the idea without using divisibility predicate,
        - $IsPrime(p) : p > 1 \wedge (\forall d \in \mathbb{N}, (\exists k \in \mathbb{Z}, p = kd) \Rightarrow d = 1 \vee d = p)$, where $p \in \mathbb{Z}$

Expressing Definitions in Programs

- Consider the divisibility predicate |, where $d \mid n$ means $\exists k \in \mathbb{Z}, n = kd$ (for $d, n \in \mathbb{Z}$)
- Without using the modulo operator %, it is challenging to translate the mathematical definition of divisibility precisely into a Python function
    - We cannot represent infinite sets in a computer program
- We can use a *property* of divisibility to restrict the set of numbers to quantify over:
    - When $n \neq 0$, every number that divides n must lie in the range $\{-|n|, -|n| + 1, \ldots, |n| - 1, |n|\}$
- In Python, we represent the above set using the range data type:
    - possible_divisors = range(- abs(n), abs(n) + 1)
- After we replace $\mathbb{Z}$ with possible_divisors, we can now translate the definition into Python code

```
def divides(d: int, n: int) -> bool:
    """Return whether d divides n."""
    possible_divisors = range(- abs(n), abs(n) + 1)
    return any({n == k * d for k in possible_divisors}))
```

- We can also translate the prime number definition in Python:

```
def is_prime(p: int) -> bool:
    """Return whether p is prime."""
    possible_divisors = range(1, p+1)
    return (
        p > 1 and
        all({d == 1 or d == p for d in possible_divisors if divides(d, p)})
    )
```

Divisibility and the Remainder Operation

- We can check whether n is divisible by 2 by checking whether then remainder when n is divided by 2 is 0 or not

```
def is_even(n: int) -> bool:
    """Return whether n is even."""
    return n % 2 == 0
```

- $\forall n, d \in \mathbb{Z}, d \neq 0 \Rightarrow (d \mid n \Leftrightarrow n \,\%\, d = 0)$
    - When $d \neq 0$, the remainder $n \,\%\, d$ is undefined
- To account for $d = 0$,

```
def divides2(d: int, n: int) -> bool:
    """Return whether d divides n."""
    if d == 0:
        return n == 0
```

```
        else:
            return n % d == 0
```

- divides2 is more *efficient* than divides though they are logically equivalent
    - Performs fewer calculations
    - No comprehensions
- To speed up our implementation of is_prime, we can call divides3 instead of divides

3.10 Testing Functions II: *hypothesis*

Property-Based Testing

- *Property-based testing* – a single test that consists of a large set of possible inputs that is generated in a programmatic way
- Property-based tests use assert statements to check for *properties* that the function being tested should satisfy
- Possible properties that every output of the function should satisfy:
    - The *type* of the output
        - The function str should always return a string
    - *Allowed values* of the output
        - The function len should always return an integer that is greater than or equal to zero
    - *Relationships* between input and output
        - The function max(x, y) should return something that is greater than or equal to both x and y
    - *Relationships* between two (or more) input-output pairs
        - "For any two lists of numbers nums1 and nums2, we know that sum(nums1 + nums2) == sum(nums1) + sum(nums2)

Using *hypothesis*

- Consider the function is_even, which we define in a file called my_functions.py

```
# Suppose we've saved this in my_functions.py

def is_even(value: int) -> bool:
    """Return whether value is divisible by 2.

    >>> is_even(2)
    True
    >>> is_even(17)
    False
```

```
    """
    return value % 2 == 0
```

- Rather than choosing specific inputs to test is_even on, we're going to test the following two *properties*:
  - is_even always return True when given an int of the form 2 * x (where x is an int)
  - is_even always return False when given an int of the form 2 * x + 1 (where x is an int)
- Using symbolic notation:
  - $\forall x \in \mathbb{Z}$, is_even$(2x)$
  - $\forall x \in \mathbb{Z}, \neg$is_even$(2x + 1)$
- To test the function, we first create a new file called test_my_functions.py and include the following "test" function

```
# In file test_my_functions.py
from my_functions import is_even

def test_is_even_2x(x: int) -> None:
    """Test that is_even returns True when given a number of the form 2 *
    x"""
    assert is_even(2 * x)
```

- The hypothesis module offers a set of *strategies* that we can use
  - These strategies are able to generate several values of a specific type of input
  - i.e. use the integers strategy to generate int data types
- We add the following 2 lines to the top of our test file:

```
from hypothesis import given
from hypothesis.strategies import integers
```

- *Decorator* – specified by using the @ symbol with an expression in the line immediately before a function definition
  - Used to attach given and integer to our test function

```
# In file test_my_functions.py
from hypothesis import given
from hypothesis.strategies import integers

from my_functions import is_even

@given(x=integers())
def test_is_even_2x(x: int) -> None:
    """Test that is_even returns True when given a number of the form 2 *
    x"""
```

```
        assert is_even(2 * x)
```

- *Integers* – a hypothesis function that returns a special data type called a *strategy*
    - *Strategy* – what hypothesis uses to generate a range of possible inputs
    - Calling integers() returns a strategy that generates ints
- *Given* – a hypothesis function that takes in arguments in the form <param>=<strategy>, which acts as a mapping for the test parameter name to a strategy that hypothesis should use for generating arguments for that parameter
- The line @given(x=integers()) *decorates* the test function, so that when we run the test function, hypothesis will call the test several times, using int values for x as specified by the strategy integers()
    - @given helps automate the process of "run the test on different int values"
- To actually run the test, we use pytest

```
if __name__ == '__main__':
        import pytest
        pytest.main(['test_my_functions.py', '-v'])
```

- Testing odd values
    - We can write multiple property-based tests in the same file and have pytest run each of them
    - This version of test_my_function.py adds a second test for numbers of the form $2x + 1$

```
# In file test_my_functions.py
from hypothesis import given
from hypothesis.strategies import integers

from my_functions import is_even

@given(x=integers())
def test_is_even_2x(x: int) -> None:
    """Test that is_even returns True when given a number of the
    form 2 * x"""
    assert is_even(2 * x)

@given(x=integers())
def test_is_even_2x_plus_1(x: int) -> None:
    """Test that is_even returns Fals when a given number of the
    form 2 * x + 1."""
    assert is_even(2 * x + 1)
```

```
if __name__ == '__main__':
    import pytest
    pytest.main(['test_my_function.py', '-v'])
```

Using *hypothesis* with Collections

- We added the following function into my_functions.py:

```
# In my_functions.py
from typing import List

def num_evens(nums: List[int]) -> int:
    """Return the number of even elements in nums."""
    return len([n for n in nums if is_even(n)])
```

- Let L_{int} be the set of lists of integers, the property we'll express is:
  - $\forall nums \in L_{int}, \forall x \in \mathbb{Z}, \text{nums\_evens}(\text{nums} + [2x]) = \text{num\_evens}(\text{nums}) + 1$
  - "For any list of integers nums and any integer x, the number of even elements of nums + [2 * x] is one more than the number of even elements of nums."
- Writing a test function,

```
# In test_my_functions.py
def test_num_evens_one_more_even(nums: List[int], x: int) -> None:
    """Test num_evens when you add one more even element."""
    assert num_evens(nums + [2 * x]) == num_evens(nums) + 1
```

- We need to use @given to tell hypothesis to generate inputs for this test function
  - i.e. @given(nums=…, x=…)
  - We can use the integers() strategy for x
  - We can import the lists function from hypothesis.strategies to create strategies for generating lists
  - We use lists(integers()) to return a strategy for generating lists of integers
- The full test file:

```
# In file test_my_functions.py
from hypothesis import given
from hypothesis.strategies import integers, lists

from my_functions import is_even, num_evens

@given(nums=lists(integers()), x=integers())
def test_num_evens_one_more_even(nums: List[int], x: int) -> None:
    """Test num_evens when you add one more even element."""
    assert num_evens(nums + [2 * x]) == num_evens(nums) + 1
```

```
if __name__ == '__main__':
    import pytest
    pytest.main(['test_my_function.py', '-v'])
```

- Choosing "enough" properties
  - A single property alone does not guarantee that a function is correct
  - The ideal goal of property-based testing is *choosing properties to verify*
    - If all of the properties are verified, then the function must be correct
  - An implementation for num_evens is correct (i.e. returns the number of even elements for any list of numbers) *if and only if* it satisfies all 3 of the following:
    - nums_evens([]) = 0
    - $\forall \text{nums} \in L_{int}, \forall x \in \mathbb{Z}, \text{nums\_evens}(\text{nums} + [2x]) = \text{nums\_evens}(\text{nums}) + 1$
    - $\forall \text{nums} \in L_{int}, \forall x \in \mathbb{Z}, \text{nums\_evens}(\text{nums} + [2x + 1]) = \text{nums\_evens}(\text{nums})$
  - This means that we can be certain that our num_evens function is correct with just 1 unit test and 2 property tests

3.11 Working with Multiple Quantifiers

Intro

- Recall the love table

	Sophia	**Thelonious**	**Stanley**	**Laura**
Breanna	False	True	True	False
Malena	False	True	True	True
Patrick	False	False	True	False
Ella	False	False	True	True

- Since the $Loves$ predicate is binary, we can quantify *both* of its inputs
- $\forall a \in A, \forall b \in B, Loves(a, b)$
  - "For every person a in A, for every person b in B, a loves b"
  - The order in which we quantified a and b doesn't matter
  - "For every person b in B, for every person a in A, a loves b" means the same thing
- The following two formulas are equivalent:
  - $\forall x \in S_1, \forall y \in S_2, P(x, y)$
  - $\forall y \in S_2, \forall x \in S_1, P(x, y)$
- The following two formulas are also equivalent:
  - $\exists x \in S_1, \exists y \in S_2, P(x, y)$

- $\exists y \in S_2, \exists x \in S_1, P(x, y)$

- The above would *not* be the case for a pair of alternating quantifiers
- $\forall a \in A, \exists b \in B, Loves(a, b)$
  - "For every person a in A, there exists a person b in B, such that a loves b"
    - "Everyone in A loves someone in B"
  - This is true: every person in A loves at least one person

a (from A)	b (a person in B who a loves)
Breanna	Thelonious
Malena	Laura
Patrick	Stanley
Ella	Stanley

  - Since the quantifier $\exists b \in B$ occurs after a, the choice of b is allowed to depend on the choice of a
- $\exists b \in B, \forall a \in A, Loves(a, b)$
  - "There exists a person b in B, where for every person a in A, a loves b"
    - "Someone in B is loved by everyone in A"
  - This is true because everyone in A loves Stanley

b (from B)	Loved by everyone in A?
Sophia	No
Thelonious	No
Stanley	Yes
Laura	No

  - Since the quantifier $\exists b \in B$ occurs before a, the choice of b must be *independent* of the choice of a
- When reading a nested quantified expression, always read from left to right and pay attention to the order of the quantifiers

<u>Translating Multiple Quantifiers into Python Code</u>

- We can represent the love table as a *list of lists* (<u>List[List[bool]]</u>)

```
# In loves.py
LOVES_TABLE = [
        [False, True, True, False],
        [False, True, True, True],
        [False, False, True, False],
        [False, False, True, True]
]
```

  - *Global constants* – variables that are *not* defined in a function definition
    - i.e. LOVES_TABLE

        - Called “global” because their *scope* is the entire Python module in which they are defined
        - Can be accessed anywhere in the file, including all function bodies
        - Can be imported and used in other Python modules
        - Available when we run the file in the Python console
- Exploring LOVES_TABLE
    - We can access LOVES_TABLE by running loves.py in the Python console

```
>>> LOVES_TABLE[0]  # The first row of the table
[False, True, True, False]
>>> LOVES_TABLE[0][1]  # The (0, 1) entry in the table
True
```

    - LOVES_TABLE[i][j] evaluates to the entry in row i and column j of the table
    - To access colomn j, we can use a list comprehension to access the j-th element in each row

```
>>> [LOVES_TABLE[i][0] for i in range(0, 4)]
[False, False, False, False]
```

    - We can add two more constants to represent the sets A and B

```
A = {
        ‘Breanna’: 0,
        ‘Malena’: 1,
        ‘Patrick’: 2,
        ‘Ella’: 3
}

B = {
        ‘Sophia’: 0,
        ‘Thelonious’: 1,
        ‘Stanley’: 2,
        ‘Laura’: 3
}
```

- We can define a loves predicate, which takes in two strings and return whether person a loves person b

```
def loves(a: str, b: str) -> bool:
        """Return whether the person at index a loves the person at index b.

        Preconditions:
        - a in A
        - b in B
```

```
>>> loves('Breanna', 'Sophia')
False
"""
a_index = A[a]
b_index = B[b]
return LOVES_TABLE[a_index][b_index]
```

- We can represent the statements in predicate logic we've written
  - $\forall a \in A, \forall b \in B, Loves(a, b)$

    ```
    >>> all ({loves(a, b) for a in A for b in B})
    False
    ```
  - $\exists a \in A, \exists b \in B, Loves(a, b)$

    ```
    >>> any ({loves(a, b) for a in A for b in B})
    True
    ```
  - $\forall a \in A, \exists b \in B, Loves(a, b)$

    ```
    >>> all({any({loves(a, b) for b in B}) for a in A})
    True
    ```
    - It is very hard to read the above statement
    - *Never nest* <u>all/any</u> *calls*
    - We can pull out the inner <u>any</u> into its own function

      ```
      def loves_someone(a: str) -> bool:
          """Return whether a loves at least one person in B.

          Preconditions:
          - a in A
          """
          return any({loves(a, b) for b in B})

      >>> all({loves_someone(a) for a in A})
      True
      ```
  - $\exists b \in B, \forall a \in A, Loves(a, b)$

    ```
    >>> any({all({loves(a, b)} for a in A) for b in B})
    ```
    - After pulling out the inner <u>all</u> expression into a named function,

      ```
      def loved_by_everyone(b: str) -> bool:
          """Return whether b is loved by everyone in A.

          Preconditions:
          - b in B
      ```

```
    """
    return all({loves(a, b)} for a in A)

>>> any({loved_by_everyone(b) for b in B})
True
```

A Famous Logical Statement

- "There are infinitely many primes."
- Because primes are natural numbers, if there are infinitely many of them, then they have to keep growing bigger and bigger
    - "Every natural number has a prime number larger than it"
    - $\forall n \in \mathbb{N}, \exists p \in \mathbb{N}, p > n \wedge IsPrime(p)$
- If we want to express this statement without either the $IsPrime$ or divisibility predicates, we would end up with an extremely cumbersome statement

4.1 Tabular Data

Toronto Getting Married

ID	Civic Centre	Marriage Licenses Issued	Time Period
1657	ET	80	Jan 2011
1658	NY	136	Jan 2011
1659	SC	159	Jan 2011
1660	TO	367	Jan 2011
1661	ET	109	Feb 2011
1662	NY	150	Feb 2011
1663	SC	154	Feb 2011
1664	TO	383	Feb 2011

- Question: what is the average number of marriage licenses issued by each civic centre?
- We can store this table as a list of lists, where each inner list represents one row of the table
- Based on the sample data we have:
    - The ids and number of marriage licenses are natural numbers
        - Use int data type
    - The civic centre is a two-letter code
        - Use str data type
    - The time periods a year-month combination
        - Use the datetime module
- In Python,

```
import datetime
marriage_data = [
    [1657, 'ET', 80, datetime.date(2011, 1, 1)],
    [1658, 'NY', 136, datetime.date(2011, 1, 1)],
    [1659, 'SC', 159, datetime.date(2011, 1, 1)],
    [1660, 'TO', 367, datetime.date(2011, 1, 1)],
    [1661, 'ET', 109, datetime.date(2011, 2, 1)],
    [1662, 'NY', 150, datetime.date(2011, 2, 1)],
    [1663, 'SC', 154, datetime.date(2011, 2, 1)],
    [1664, 'TO', 383, datetime.date(2011, 2, 1)]
]
```

- We can play around with the data

```
>>> len(marriage_data)  # There are 8 rows of data
8
>>> len(marriage_data[0]  # The first rown has 4 elements
4
>>> [len(row) for row in marriage_data]  # Every row has 4 elements
[4, 4, 4, 4, 4, 4, 4, 4]
>>> marriage_data[0]
[1657, 'ET', 80, datetime.date(2011, 1, 1)]
>>> marriage_data[1]
[1658, 'NY', 136, datetime.date(2011, 1, 1)]
>>> marriage_data[0][0]
1657
>>> marriage_data[0][1]
'ET'
>>> marriage_data[0][2]
80
>>> marriage_data[0][3]
datetime.date(2011, 1, 1)
```

<u>Accessing Columns and Filtering Rows</u>

- We can retrieve a column by using a list comprehension

```
>>> [row[1] for row in marriage_data]  # The civic centre column
['ET', 'NY', 'SC', 'TO', 'ET', 'NY', 'SC', 'TO']
>>> {row[1] for row in marriage_data}
{'NY', 'TO', 'ET', 'SC'}
```

- We can retrieve all rows corresponding to a specific civic centre using if conditions in comprehensions

```
>>> [row for row in marriage_data if row[1] == 'TO']
[[1660, 'TO', 367, datetime.date(2011, 1, 1)], [1664, 'TO', 383,
datetime.date(2011, 2, 1)]]
```

- We can also filter rows based on a threshold for the number of marriage licenses issued

```
>>> [row for row in marriage_data if row[2] > 380]
[[1664, 'TO', 383, datetime.date(2011, 2, 1)]]
```

A Worked Example

- Question: what is the average number of marriage licenses issued by each civic centre?
- To solve the question, we need to create a dictionary comprehension

```
>>> names = {row[1] for row in marriage_data}
>>> names
{'NY', 'TO', 'ET', 'SC'}
>>> {key: 0 for key in names}
{'NY': 0, 'TO': 0, 'ET': 0, 'SC': 0}
```

- Calculate the average number of marriage licenses issued per month for the 'TO' civic centre

```
>>> [row for row in marriage_data if row[1] == 'TO']  # The 'TO' rows
[[1660, 'TO', 367, datetime.date(2011, 1, 1)], [1664, 'TO', 383,
datetime.date(2011, 2, 1)]]
>>> [row[2] for row in marriage_data if row[1] == 'TO']  # The 'TO' marriages
issued
[367, 383]
>>> issued_by_TO = [row[2] for row in marriage_data if row[1] == 'TO']
```

- Now issued_by_TO is a list containing the number of marriage licenses issued by the 'TO' civic centre
- We can calculate its average by dividing its length

```
>>> sum(issued_by_TO) / len(issued_by_TO)
375.0
```

- To merge the above code with the dictionary comprehension outline, first design a function that calculates the average for only 1 civic centre

```
from typing import List

def average_licenses_issued(data: List[list], civic_centre: str) -> float:
    """Return the average number of marriage licenses issued by civic_centre
    in data.
```

```
    Return 0.0 if civic_centre does not appear in the given data.

    Preconditions:
    - all({len(row) == 4 for row in data})
    - data is in the format described in Section 4.1
    """
    issued_by_civic_centre = [row[2] for row in data if row[1] == civic_centre]

    if issued_by_civic_centre == []:
        return 0.0
    else:
        total = sum(issued_by_civic_centre)
        count = len(issued_by_civic_centre)

        return total / count
```

```
>>> average_licenses_issued(marriage_data, 'TO')
375.0
```

- Finally, we can combine it with our dictionary comprehension ('TO' can be replaced with the key that is changing)

```
>>> {key: 0 for key in names}
{'NY': 0, 'TO': 0, 'ET': 0, 'SC': 0}
>>> {key: average_licences_issued(marriage_data, key) for key in names}
{'NY': 143.0, 'TO': 375.0, 'ET': 94.5, 'SC': 156.5}
```

- We will save our work by writing the entirety of above as a function:

```
def average_licenses_by_centre(marriage_data: List[list]) -> Dict[str, float]:
    """Return a mapping of the average number of marriage licenses issued
    at each civic centre.

    In the returned mapping:
    - Each key is the name of a civic centre
    - Each corresponding value is the average number of marriage licences
    issued at that centre.

    Preconditions:
    - marriage_data is in the format described in Section 4.1
    """
```

```
    names = {'TO', 'NY', 'ET', 'SC'}
    return {key: average_licenses_issued(marriage_data, key) for key in
    names}
```

4.2 Defining Our Own Data Types, Part 1

Defining a Data Class

- *Class* – another term for data type
    - type(3) evaluates to <class 'int'>
- *Data class* – the simplest kind of data type
    - Purpose: to bundle individual pieces of data into a single Python object
- If we want to represent a "person" consisting of a given name, family name, age, and home address, we could define our own data class to create a new data type consisting of these four values:

```
from dataclasses import dataclass


@dataclass
class Person:
    """A custom data type that represents data for a person."""
    given_name: str
    family_name: str
    age: int
    address: str
```

    - from dataclasses import dataclass is a Python import statement that lets us use dataclass below
    - @dataclass is a Python *decorator*. It tells Python that the data type we're defining is a data class
    - class Person: signals the start of a *class definition*
        - The name of the class is Person
        - The rest of the code is indented to put it inside of the class body
    - The next line is a docstring that describes the purpose of the class
    - Each remaining line defines a piece of data associated with the class
        - Each piece of data is called an *instance attribute* of the class
        - For each instance attribute, we write a name and a type annotation
- General data class definition syntax

```
@dataclass
class <ClassName>:
    """Description of data class.
```

“””
<attribute1>: <type1>
<attribute2>: <type2>
…

Using Data Classes

- To create an instance of our Person data class, we write a Python expression that calls the data class, passing in as arguments of the values for each instance attribute

```
>>> david = Person('David', 'Liu', 100, '40 St. George Street')
>>> type(david)
<class Person>
```

- To access the individual values, we use the names of the instance attributes together with *dot notation*

```
>>> david.given_name
'David'
>>> david.family_name
'Liu"
>>> david.age
100
>>> david.address
'40 St. George Street'
```

- Much more readable tan list indexing

Tip: Naming Attributes when Creating Data Class Instances

- In the expression Person('David', 'Liu', 100, '40 St. George Street'), the order of the arguments must match the order the instance attributes are listed in the definition of the data class
- Python enables us to create data class instances using *keyword arguments* to explicitly name which argument corresponds to which instance attribute:

```
david = Person(given_name = 'David', family_name = 'Liu', age = 100, address =
'40 St. George Street')
```

- Using keyword arguments allows us to pass the values in any order we want:

```
david = Person(family_name = 'Liu', given_name = 'David', address = '40 St.
George Street', age = 100)
```

- Improves readability of code
- Representing data classes in the memory model

Variable Value
david

```
Person(
        family_name = 'Liu',
        given_name = 'David',
        address = '40 St. George Street'
        age = 100
)
```

4.3 Defining Our Own Data Types, Part 2

Constraining Data Class Values: Representation Invariants

- We don't always want to allow every possible value of a given type for an attribute value
    - i.e. we don't want negative numbers as age
- We can record a second piece of information about the age attribute: age >= 0
    - This kind of constraint is called a *representation invariant*
        - *Representation invariant* – a predicate describing a condition on how we *represent* something that must always be true
- All type annotations, like age: int, are representation invariants
- To add non-type-annotation representation invariants,

```
@dataclass
class Person:
    """A custom data type that represents data for a person.

    Representation Invariants:
    - self.age >= 0
    """
    given_name: str
    family_name: str
    age: int
    address: str
```

- In the class docstring, we use the variable name self to refer to a generic instance of the data class
    - This use of self is Python convention
- Checking representation invariants automatically with python_ta
    - Like preconditions, representation invariants are *assumptions* that we make about values of a data type
        - i.e. we can assume that every Person instance has an age that's greater than or equal to zero

- Representation invariants are also *constraints* on how we can create a data class instance
- python_ta.contracts supports checking all representation invariants:

```
# class Person above

if __name__ == '__main__':
    import python_ta.contracts
    python_ta.contracts.DEBUG_CONTRACTS = False
    python_ta.contracts.check_all_contracts()
```

- Then, a negative age would output an assertion error

The Data Class Design Recipe

- 1. Write the class header

```
@dataclass
class Person:
```

  - The class header consists of 3 parts:
    - The @dataclass decorator
    - The keyword class
    - The name of the data class
  - Remember to import from dataclasses
  - The name of the class should use the "CamelCase" naming convention
    - i.e. capitalize every word, *not* separate the words with underscores
- 2. Write the instance attributes for the data class

```
@dataclass
class Person:
    given_name: str
    family_name: str
    age: int
    address: str
```

  - Every instance of the data class will have *all* of these attributes
  - The name of the attributes should use the "snake_case" naming convention
    - Like functions and variable names
  - Use the typing module whenever possible
- 3. Write the data class docstring

```
@dataclass
class Person:
    """A data class representing a person.
```

Instance Attributes:
- given_name: the person's given name
- family_name: the person's family name
- age: the person's age
- address: the person's address
"""
given_name: str
family_name: str
age: int
address: str

- Write a description of the class and a description for every instance attribute
- Use the header "Instance Attributes:" to mark the beginning of the attribute descriptions

- 4. Write an example instance (optional)

@dataclass
class Person:
"""A data class representing a person.

Instance Attributes:
- given_name: the person's given name
- family_name: the person's family name
- age: the person's age
- address: the person's address

>>> david = Person('David', 'Liu', 40, '40 St. George Street')
"""
given_name: str
family_name: str
age: int
address: str

- Used to illustrate all of the instance attributes

- 5. Document any additional representation invariants

@dataclass
class Person:
"""A data class representing a person.

Instance Attributes:
- given_name: the person's given name

```
    - family_name: the person's family name
    - age: the person's age
    - address: the person's address

    Representation Invariants:
    - self.age >= 0

    >>> david = Person('David', 'Liu', 40, '40 St. George Street')
    """
    given_name: str
    family_name: str
    age: int
    address: str
```

- Each representation invariant should be a boolean expression in Python
- Use self.<attribute> to refer to an instance attribute within a representation invariant

A Worked Example

- Recall the marriage license data set

```
marriage_data = [
    [1657, 'ET', 80, datetime.date(2011, 1, 1)],
    [1658, 'NY', 136, datetime.date(2011, 1, 1)],
    [1659, 'SC', 159, datetime.date(2011, 1, 1)],
    [1660, 'TO', 367, datetime.date(2011, 1, 1)],
    [1661, 'ET', 109, datetime.date(2011, 2, 1)],
    [1662, 'NY', 150, datetime.date(2011, 2, 1)],
    [1663, 'SC', 154, datetime.date(2011, 2, 1)],
    [1664, 'TO', 383, datetime.date(2011, 2, 1)]
]
```

- Rather than storing each row in the table as a list, we can introduce a new data class to store this information

```
from dataclasses import dataclass
from datetime import date

@dataclass
class MarriageData
    """A record of the number of marriage licenses issued in a civic centre in
    a given month.
```

Instance Attributes:
- id: a unique identifier for the record
- civic_centre: the name of the civic centre
- num_licenses: the number of licenses issued
- month: the month these licenses were issued
“””
id: int
civic_centre: str
num_licenses: int
month: date

- Using the above data class, we can represent tabular data as a list of MarriageData instances rather than a list of lists
- The values representing each entry in the table are the same, but how we “bundle” each row of data into a single entity is different:

```
>>> marriage_data = [
        MarriageData(1657, ‘ET’, 80, datetime.date(2011, 1, 1)),
        MarriageData(1658, ‘NY’, 136, datetime.date(2011, 1, 1)),
        MarriageData(1659, ‘SC’, 159, datetime.date(2011, 1, 1)),
        MarriageData(1660, ‘TO’, 367, datetime.date(2011, 1, 1)),
        MarriageData(1661, ‘ET’, 109, datetime.date(2011, 2, 1)),
        MarriageData(1662, ‘NY’, 150, datetime.date(2011, 2, 1)),
        MarriageData(1663, ‘SC’, 154, datetime.date(2011, 2, 1)),
        MarriageData(1664, ‘TO’, 383, datetime.date(2011, 2, 1))
]
```

- Instead of writing row[1] and row[2] in a comprehension, we now write row.civic_centre and row.num_licenses
    - This is more explicit in what attributes of the data are accessed
    - “*Explicit is better than implicit*”

4.4 Repeated Execution: For Loops

Introducing the Problem: Repeating Code

- *Variable reassignment statement* – assigning a value to a variable that has *already* been given a value
    - i.e. sum_so_far = sum_so_far + numbers[0]

- First, the right-hand side of the assignment (sum_so_far + numbers[0]) statement is evaluated; then, the value is stored in the variable on the left-hand side (sum_so_far)

The *For* Loop

- *For loop* – a compound statement that repeats a block of code once for element in a collection
- Syntax of a for loop:

```
for <loop_variable> in <collection>:
    <body>
```

- Parts of a for loop:
  - <collection> is an expression for a Python collection (e.g. a list or set)
  - <loop_variable> is the name for the *loop variable* that will refer to an element in the collection
  - <body> is a sequence of one or more statements that will be repeatedly executed.
    - Called the *body* of the for loop
    - The statements within the loop body may refer to the loop variable to access the "current" element in the collection
  - The body of a for loop *must* be indented relative to the for keyword
- When a for loop is executed,
  - 1. The loop variable is assigned to the first element in the collection
  - 2. The loop body is executed, using the current value of the loop variable
  - 3. Steps 1 and 2 repeat for the second element of the collection, then the third, etc. until all elements of the collection have been assigned to the loop variable exactly once

The Accumulator Pattern and *my_sum*

- Implementation of my_sum

```
def my_sum(numbers: List[int]) -> int:
    """Return the sum of the given numbers.

    >>> my_sum([10, 20, 30])
    60
    """
    sum_so_far = 0

    for number in numbers:
```

```
        sum_so_far = sum_so_far + number

    return sum_so_far
```

- We no longer need to use list indexing (numbers[n]) to access individual list elements
    - The for loop in Python handles the extracting of individual elements for us
- Accumulators and tracing through loops
    - The frequent reassignment of a variable can make loops hard to reason about
    - We call the variable sum_so_far the *loop accumulator*
    - *Loop accumulator* – stores an aggregated result based on the elements of the collection that have been previously visited by the loop
    - We can keep track of the execution of the different iterations of the loop in the *loop accumulation table* consisting of three columns:
        - How many iterations have occurred so far
        - the value of the loop variable for that iteration
        - The value of the loop accumulator at the *end* of that iteration

Iteration	Loop variable (number)	Loop accumulator (sum_so_far)
0	N/A	0
1	10	10
2	20	30
3	30	60

    - Use the _so_far suffix in the variable name of accumulator variables and add a comment explaining the purpose of the variable

```
def my_sum(numbers: List[int]) -> int:
    """Return the sum of the given numbers.

    >>> my_sum([10, 20, 30])
    60
    """
    # ACCUMULATOR sum_so_far: keep track of the running sum of the
    elements in numbers
    sum_so_far = 0

    for number in numbers:
        sum_so_far = sum_so_far + number

    return sum_so_far
```

- When the collection is empty
    - When we call my_sum on an empty list,

```
>>> my_sum([])
0
```

- This happens because sum_so_far is assigned to 0, and then the for loop does not execute any code, and do 0 is returned
- *When the collection is empty, the initial value of* sum_so_far *is returned*

Designing Loops Using the Accumulator Pattern

- The *accumulator pattern*
  - 1. Choose a meaningful name for an accumulator variable based on what we're computing
    - Use the suffix _so_far to remind ourselves that this is an accumulator
  - 2. Pick an initial value for the accumulator
    - This value is usually what should be returned if the collection is empty
  - 3. In the loop body, update the accumulator variable based on the current value of the loop variable
  - 4. After the loop ends, return the accumulator

```
<x>_so_far = <default_value>

for element in <collection>:
    <x>_so_far = … <x>_so_far … element …  # Somehow combine loop variable and accumulator

return <x>_so_far
```

- Accumulating the product

```
from typing import List

def product(numbers: List[int]) -> int:
    """Return the product of the given numbers.

    >>> product([10, 20])
    200
    >>> product([-5, 4])
    -20
    # ACCUMULATOR product_so_far: keep track of the product of the
    # elements in numbers seen so far in the loop
    product_so_far = 1

    for number in numbers:
```

```
        product_so_far = product_so_far * number

    return product_so_far
```

Looping Over Sets

- Because sets are unordered, we cannot assume a particular order that the for loop will visit the elements in
- For loops over sets should be used when *the same result would be obtained regardless of the order of the elements*
    - i.e. sum

Looping Over Strings

- Python treats a string as an ordered collection of characters (strings of length one)
- We can use for loops with strings to iterate over its characters one at a time
- Example:

```
def my_len(s: str) -> int:
    """Return the number of characters in s

    >>> my_len('David')
    5
    """
    # ACCUMULATOR len_so_far: keep track of the number of
    # characters in s seen so far in the loop
    len_so_far = 0

    for _ in s:
        len_so_far = len_so_far + 1

    return len_so_far
```

    - We used an underscore here because we don't care what the actual value the character is – we are only counting iterations
        - The loop variable is not used in the body of the for loop

Looping Over Dictionaries

- When we iterate over a dictionary, the loop variable refers to the *key* of each key-value pair
- For example, we are given a dictionary mapping restaurant menu items (as strings) to their prices (as floats) and we want to calculate the sum of all the prices on the menu:

```
def total_menu_price(menu: Dict[str, float]) -> float:
    """Return the total price of the given menu items.

    >>> total_menu_price({'fries': 3.5, 'hamburger': 6.5})
    10.0
    """
    # ACCUMULATOR total_so_far: keep track of the total cost of
    # all items in the menu seen so far in the loop.
    total_so_far = 0.0

    for item in menu:
        total_so_far = total_so_far + menu[item]

    return total_so_far
```

- Loop accumulation table:

Iteration	Loop variable (item)	Loop accumulator (total_so_far)
0		0.0
1	'fries'	6.5
2	'hamburger'	10.0

- Like sets, dictionaries are unordered

A New Type Annotation: *Iterable*

- my_len also works on lists, sets, and other collections
  - The type annotation does not have to be restricted to strings
- A Python data type is *iterable* when its values can be used as the "collection" of a for loop
  - A Python object is iterable when it is an instance of an iterable data type
- Equivalent to when a value can be used as the "collection" of a comprehension
- Import the Iterable type from typing
- A more general my_len:

```
from typing import Iterable

def my_len(collection: Iterable) -> int:
    """Return the number of characters in collection

    >>> my_len('David')
    5
    >>> my_len([1, 2, 3])
```

```
    3
    >>> my_len({'a': 1000})
    1
    """
    # ACCUMULATOR len_so_far: keep track of the number of
    # characters in s seen so far in the loop
    len_so_far = 0

    for _ in collection:
        len_so_far = len_so_far + 1

    return len_so_far
```

- Accumulators can work with any iterable object
- Alternatives to for loops
    - Many of the above examples can be solved using comprehensions rather than loops
    - Comprehensions are often shorter and more direct translations of a computation than for loops
    - For loops allow us to customize exactly how filtering and aggregation occurs

4.5 For Loop Variations

<u>Multiple Accumulators</u>

- For example, given a dictionary mapping menu items to prices, we can get the average price using 2 accumulators:

```
def average_menu_price(menu: Dict[str, float]) -> float:
    """Return the average price of an item from the menu.

    >>> average_menu_price({'fries': 3.5, 'hamburger': 6.5})
    5.0
    """
    # ACCUMULATOR len_so_far: keep track of the number of
    # items in the menu seen so far in the loop
    len_so_far = 0
    # ACCUMULATOR total_so_far: keep track of the cost of
    # all items in the menu seen so far in the loop
    total_so_far = 0.0
```

```
    for item in menu:
        len_so_far = len_so_far + 1
        total_so_far = total_so_far + menu[item]

    return total_so_far / len_so_far
```

- Loop accumulation table:

Iteration	Loop variable (item)	Accumulator len_so_far	Accumulator total_so_far
0		0	0.0
1	'fries'	1	6.5
2	'hamburger'	2	10.0

Conditional Execution of the Accumulator

- Consider the following problem: given a string, count the number of vowels in the string
- By nesting an if statement inside a for loop, we can adapt our accumulator pattern to only update the accumulator when certain conditions are met

```
def count_vowels(s: str) -> int:
    """Return the number of vowels in s.

    >>> count_vowels('aeiou')
    5
    >>> count_vowels('David')
    2
    """
    # ACCUMULATOR vowels_so_far: keep track of the number of vowels
    # seen so far in the loop.
    vowels_so_far = 0

    for letter in s:
        if letter in 'aeiou':
            vowels_so_far = vowels_so_far + 1

    return vowels_so_far
```

- If the word is the empty string, the for loop will not iterate once and the value 0 is returned
- Two cases for the loop body:
  - 1. When letter is a vowel, the reassignment vowels_so_far = vowels_so_far + 1 increases the number of vowels seen so far by 1

        - 2. When letter is not a vowel, nothing else happens in the current iteration because this if statement has no else branch.
            - The vowel count remains the same
    - Loop accumulation table for count_vowels('David')

Loop Iteration	Loop Variable letter	Accumulator vowels_so_far
0		0
1	'D'	0
2	'a'	1
3	'v'	1
4	'i'	2
5	'd'	2

    - This function can be compared to an equivalent implementation using a filtering comprehension
- Implementing max

```
from typing import List

def my_max(numbers: List[int]) -> int:
    """Return the maximum value of the numbers in numbers.

    Preconditions:
    - numbers != []

    >>> my_max([10, 20])
    20
    >>> my_max([-5, -4])
    -4
    """
    # ACCUMULATOR max_so_far: keep track of the maximum value
    # of the elements in numbers seen so far in the loop.
    max_so_far = numbers[0]

    for number in numbers:
        if number > max_so_far:
            max_so_far = number

    return max_so_far
```

    - The accumulator max_so_far is updated only when a larger number is encountered (if number > max_so_far)

- Existential search

```
def starts_with(strings: Iterable[str], char: str) -> bool:
    """Return whether one of the given strings starts with the character char.

    Precondition:
    - all({s != '' for s in strings})
    - len(char) == 1

    >>> starts_with(['Hello', 'Goodbye', 'David', 'Mario'], 'D')
    True
    >>> starts_with(['Hello', 'Goodbye', 'David', 'Mario'], 'A')
    False
    """
    return any({s[0] == char for s in words})
```

  - To implement this function *without* using the any function, consider the following:
    - The syntax for s in words can be used to create a for loop
    - The expression s[0] == char can be used as a condition for an if statement

```
def starts_with_accumulator(words: Iterable[str], char: str) -> bool:
    """..."""
    # ACCUMULATOR starts_with_so_far: keep track of whether
    # any of the words seen by the loop so far starts with char
    starts_with_so_far = False

    for s in strings:
        if s[0] == char:
            starts_with_so_far = True

    return starts_with_so_far
```

  - To update the accumulator, we set it to True when the current string s starts with char
  - Loop accumulation table:

Iteration	Loop variable s	Accumulator starts_with_so_far
0		False
1	'Hello'	False
2	'Goodbye'	False
3	'David'	True
4	'Mario'	True

- Early returns
    - The function starts_with_v2 performs unnecessary work because it must loop through every element of the collection before returning a result
    - As soon as condition s[0] == char evaluates to True, we know that the answer is *yes* without checking any of the remaining strings
    - We can use the return statement inside the body of the loop
    - No code execute after the return statement

```
def starts_with_early_return(strings: Iterable[str], char: str) -> bool:
    """..."""
    for s in strings:
        if s[0] == char:
            return True

    return False
```

        - We no longer have the accumulator variable
- One common error

```
def starts_with_wrong(strings: Iterable[str], char: str) -> bool:
    """..."""
    for s in strings:
        if s[0] == char:
            return True
        else:
            return False
```

    - The loop will only ever perform 1 iteration
    - Existential searches are asymmetric:
        - The function can return True early as soon as it has found an element of the collection meeting the desired criterion
        - To return False, it must check *every* element of the collection
- Universal search

```
def all_start_with(strings: Iterable[str], char: str) -> bool:
    """Return whether all of the given strings starts with the character char.

    Precondition:
    - all({s != '' for s in strings})
    - len(char) == 1

    >>> all_starts_with(['Hello', 'Goodbye', 'David', 'Mario'], 'D')
    False
```

```
        >>> all_starts_with(['Drip', 'Drop', 'Dangle'], 'D')
        True
        """
        return all({s[0] == char for s in strings})
```

- We can use the accumulator pattern from starts_with_accumulator
- A better way is to write this function using an early return, since we can return False as soon as a counterexample is found:

```
def all_starts_with_early_return(strings: Iterable[str], char: str) -> bool:
    """..."""
    for s in words:
        if s[0] != char:
            return False

    return True
```

4.6 Index-Based For Loops

Repeating Code

- Recall the my_sum function

```
def my_sum(numbers: List[int]) -> int:
    """Return the sum of the given numbers.

    >>> my_sum([10, 20, 30])
    60
    """
    # ACCUMULATOR sum_so_far: keep track of the running sum of the
    elements in numbers
    sum_so_far = 0

    for number in numbers:
        sum_so_far = sum_so_far + number

    return sum_so_far
```

- For the my_sum function, we know that the index starts at 0 and ends at the length – 1

```
def my_sum_v2(numbers: List[int]) -> int:
    """..."""
    # ACCUMULATOR sum_so_far: keep track of the running sum of the
    # elements in numbers.
```

```
    sum_so_far = 0

    for i in range(0, len(numbers)):
        sum_so_far = sum_so_far + numbers[i]

    return sum_so_far
```

- Differences between my_sum and my_sum_v2:
  - Loop variable number vs. i:
    - number refers to an element of lthe list numbers (starting with the first element)
    - i refers to an integer (starting at 0)
  - Looping over a list vs. a range:
    - for number in numbers causes the loop body to execute once for each element in numbers
    - for i in range(0, len(numbers)) causes the loop body to execute once for each integer in range(0, len(numbers))
  - Updating the accumulator:
    - Since number refers to a list element, we can add it directly to the accumulator
    - Since i refers to *where* we are in the list, we access the corresponding list element using list indexing to add it to the accumulator
- Both our element-based and index-based implementations are correct here

When Location Matters

- Example

```
def count_adjacent_repeats(string: str) -> int:
    """Return the number of times in the given string that two adjacent characters
    are equal

    >>> count_adjacent_repeats('look')
    1
    >>> count_adjacent_repeats('David')
    0
    """
```

- We want to use an accumulator variable that starts at 0 and increases by 1 every time two adjacent repeated characters are found
- Comparisons:
  - string[0] == string[1]

- string[1] == string[2]
- etc.

```
def count_adjacent_repeats(string: str) -> int:
    """..."""
    # ACCUMULATOR repeats_so_far: keep track of the number of adjacent
    # characters that are identical
    repeats_so_far = 0

    for i in range(0, len(string) - 1):
        if string[i] == string[i + 1]:
            repeats_so_far = repeats_so_far + 1

    return repeats_so_far
```

- Since we are indexing string[i + 1], our loop variable i only needs to go up to n – 2 rather than n – 1

- We could not have implemented the above function using an element-based for loop
  - We would not be able to access the character adjacent the current one

Two Lists, One Loop

- Index-based for loops can be used to iterate over two collections in parallel using a single for loop
- Example

```
def count_money(counts: List[int], denoms: List[float]) -> float:
    """Return the total amount of money for the given coin counts and
    denominations.

    counts stores the number of coins of each type, and denominations stores the
    value of each coin type. Each element in counts corresponds to the element at
    the same index in denoms.

    Preconditions:
    - len(counts) == len(values)

    >>> count_money([2, 4, 3], [0.05, 0.10, 0.25])
    1.25
    """
```

- We need to multiply each corresponding element of counts and denoms, and add the results:

    - (counts[0] * denoms[0]) + (counts[1] * denoms[1]) + …
- Using range,
    - [counts[i] * denoms[i] for i in range(0, len(counts))]
- Then, we can compute the sum sing the built-in Python function

```
def count_money(counts: List[int], denoms: List[float]) -> float:
    """…"""
    return sum([counts[i] * denoms[i] for i in range(0, len(counts))])
```

- The above works well, however, we can also implement the function using a for loop

```
def count_money(counts: List[int], denoms: List[float]) -> float:
    """…"""
    # ACCUMULATOR money_so_far: keep track of the total money so far.
    money_so_far = 0.0

    for i in range(0, len(counts)):
        money_so_far = money_so_far + counts[i] * values[i]

    return money_so_far
```

Choosing the Right For Loop

- Element-based for loops (for <loop_variable> in <collection>) are useful when:
    - We want to process each element in the collection without knowing about its position in the collection
- Index-based for loops (for <loop_variable> in <range>) are useful when:
    - The location of elements in the collection matters
        - As in count_adjacent_repeats
    - We want to loop through more than 1 list at a time using the same index for both lists
        - As in count_money
- Though index-based loops are more powerful than element-based loops
    - Not all collections can be indexed (i.e. set and dict)
    - Index-based for loops are more *indirect* than element-based for loops
- Use element-based for loops if possible

4.7 Nested For Loops

Nested Loops and Nested Data

- Suppose we have a list of lists of integers
    - >>> lists_of_numbers = [[1, 2, 3], [10, -5], [100]]

- Our goal is to compute the sum of all the elements of the list

```
def sum_all(lists_of_numbers: List[List[int]]) -> int:
    """Return the sum of all the numbers in the given lists_of_numbers.

    >>> sum_all([[1, 2, 3], [10, -5], [100]])
    111
    # ACCUMULATOR sum_so_are: keep track of the running sum of the numbers.
    sum_so_far = 0

    for numbers in lists_of_numbers:  # numbers is a list of numbers, not 1 number
        sum_so_far = sum_so_far + sum(numbers)

    return sum_so_far
```

- *Without* using sum, we need another for loop:

```
def sum_all(lists_of_numbers: List[List[int]]) -> int:
    """…"""
    # ACCUMULATOR sum_so_far: keep track of the running sum of the numbers.
    sum_so_far = 0

    for numbers in lists_of_numbers:  # numbers is a list of numbers, not 1 number
        for number in numbers:  # number is a single number
            sum_so_far = sum_so_far + numbers

    return sum_so_far
```

  - for number in numbers loops is *nested* within the for numbers in lists_of_numbers
  - If we call our doctest example, sum_all([[1, 2, 3], [10, -5], [100]]), the following happens:
    - 1. The assignment statement sum_so_far = 0 execute, creating our accumulator variable
    - 2. The outer loop is reached
      - The loop variable list_of_numbers is assigned the first element in lists_of_numbers, which is [1, 2, 3]
      - Then, the body of the outer loop is executed. Its body is just 1 statement: the inner for loop, for number in numbers
        - The inner loop variable number is assigned the first value in numbers, which is 1

        - The inner loop body gets executed, updating the accumulator. sum_so_far is reassigned to 1 (since 0 + 1 == 1)
        - The inner loop iterates twice more, for number = 2 and number_3. At each iteration, the accumulator is updated, first by adding 2 and then 3. At this point, sum_so_far = 6 (which is 0 + 1 + 2 + 3)
        - After all 3 iterations of the inner loop occur, the inner loop stops. The Python interpreter is done executing this statement.
    - The next iteration of the *outer loop* occurs; numbers is assigned to the list [10, -5]
    - Again, the body of the outer loop occurs
        - The inner loop now iterates twice: for number = 10 and number = -5. sum_so_far is reassigned twice more, with a final value of 11 (which is 6 + 10 + -5)
    - The outer loop iterates one more time, for numbers = [100]
    - Again, the body of the outer loop occurs
        - The inner loop iterates once, for number = 100. sum_so_far is reassigned to 111 (which is 11 + 100)
    - At last, there are no more iterations of the outer loop, and so it stops
- 3. After the outer loop is done, the return statement executes, returning the value of sum_so_far, which is 111

- The above behaviour can be summarized by a *loop accumulation table*

Outer Loop Iteration	Outer Loop Variable (list_of_numbers)	Inner Loop Iteration	Inner Loop Variable (number)	Accumulator (sum_so_far)
0				0
1	[1, 2, 3]	0	1	0
1	[1, 2, 3]	1	2	1
1	[1, 2, 3]	2	3	3
1	[1, 2, 3]	3		6
2	[10, -5]	0	10	6
2	[10, -5]	1	-5	16
2	[10, -5]	2		11

3	[100]	0	100	11
3	[100]	1		111

The Cartesian Product

- We want to use nested loops on two different collections, obtaining all pairs of possible values from each collection

```
def product(set1: set, set2: set) -> Set[tuple]:
    """Return the Cartesian product of set1 and set2.

    >>> result = product({10, 11}, {5, 6, 7})
    >>> result == {(10, 5), (10, 6), (10, 7), (11, 5), (11, 6), (11, 7)}
    True
    """
    # ACCUMULATOR product_so_far: keep track of the tuples from the pairs
    # of elements visited so far.
    product_so_far = set()

    for x in set1:
        for y in set2:
            product_so_far = set.union(product_so_far, {(x, y)}_

    return product_so_far
```

  - Loop accumulation table

Outer Loop Iteration	Outer Loop Var (x)	Inner Loop Iteration	Inner Loop Var (y)	Accumulator (product_so_far)
0				set()
1	10	0		set()
1	10	1	5	{(10, 5)}
1	10	2	6	{(10, 5), (10, 6)}
1	10	3	7	{(10, 5), (10, 6), (10, 7)}
2	11	0		{(10, 5), (10, 6), (10, 7)}
2	11	1	5	{(10, 5), (10, 6), (10, 7), (11, 5)}
2	11	2	6	{(10, 5), (10, 6), (10, 7), (11, 5), (11, 6)}
2	11	3	7	{(10, 5), (10, 6), (10, 7), (11, 5), (11, 6), (11, 7)}

Outer and Inner Accumulators

- *Each loop* can have its own accumulator
- Example: suppose we have a list of lists of integers called grades

```
grades = [
    [70, 75, 80],  # ENG196
    [70, 80, 90, 100],  # CSC110
    [80, 100]  # MAT137
]
```

  - Each element of grades corresponds to a course and contains a list of grades obtained in that course
  - The list of grades for course ENG196 does not have the same length as CSC110 or MAT137
- We have defined a function average that calculates the average to a list of int
- Goal: return a new list containing the *average grade* of each course
- We can calculate a list of averages for each course using a comprehension:

```
def course_averages_comprehension(grades: List[List[int]]) -> List[float]
    """Return a new list for which each element is the average of the grades in the
    inner list at the corresponding position of grades.

    >>> course_averages_comprehension([[70, 75, 80], [70, 80, 90, 100], [80, 100]])
    [75.0, 85.0, 90.0]
    """
    return [average(course_grades) for course_grades in grades]
```

- We can translate the above function into a for loop using a list accumulator variable and list concatenation for the update

```
def course_averages_loop(grades: List[List[int]]) -> List[float]
    """..."""
    # ACCUMULATOR averages_so_far: keep track of the averages of the lists
    # visited so far in grades
    average_so_far = []

    for course_grades in grades:
        course_average = average(course_grades)
        averages_so_far = averages_so_far + [course_average]

    return averages_so_far
```

- We can also implement the function without using the average function by *expanding the definition of* average directly in the loop body

```
def course_averages(grades: List[List[int]]) -> List[float]
    """..."""
    # ACCUMULATOR averages_so_far: keep track of the averages of the lists
    # visited so far in grades
    average_so_far = []

    for course_grades in grades:
        # ACCUMULATOR len_so_far: keep track of the number of elements
        # seen so far in course_grades
        len_so_far = 0
        # ACCUMULATOR total_so_far: keep track of the total of the elements
        # seen so far in course_grades
        total_so_far = 0

        for grade in course_grades:
            len_so_far = len_so_far + 1
            total_so_far = total_so_far + grade

        course_average = total_so_far / len_so_far

        average_so_far = averages_so_far + [course_average]

    return averages_so_far
```

- The inner loop accumulators are assigned to *inside* the body of the outer loop rather than at the top of the function body
    - This is because len_so_far and total_so_far are specific to course_grades, which changes at each iteration of the outer loop
    - The statements len_so_far = 0 and total_so_far = 0 act to "reset" these accumulators for each new course_grades list
- Loop accumulation table

Outer Loop Iteration	Outer Loop Variable (course_grades)	Inner Loop Iteration	Inner Loop Variable (grade)	Inner Accumulator (len_so_far)	Inner Accumulator (total_so_far)	Outer Accumulator (averages_so_far)
0						[]
1	[70, 75, 80]	0		0	0	[]
1	[70, 75, 80]	1	70	1	70	[]

1	[70, 75, 80]	2	75	2	145	[]
1	[70, 75, 80]	3	80	3	225	[75.0]
2	[70, 80, 90, 100]	0		0	0	[75.0]
2	[70, 80, 90, 100]	1	70	1	70	[75.0]
2	[70, 80, 90, 100]	2	80	2	150	[75.0]
2	[70, 80, 90, 100]	3	90	3	240	[75.0]
2	[70, 80, 90, 100]	4	100	4	340	[75.0, 85.0]
3	[80, 100]	0		0	0	[75.0, 85.0]
3	[80, 100]	1	80	1	80	[75.0, 85.0]
3	[80, 100]	2	100	2	180	[75.0, 85.0, 90.0]

Summary: Understanding and Simplifying Nested For Loops

- Follow the following guidelines when using nested loops
    - Use nested loops when we have a single accumulator that can be initialized just once before the nested loop (e.g. sum_all and cartesian_product)
    - If we have a nested loop where the inner loop can be replaced by a built-in aggregation function (e.g. sum or len), use the built-in function instead
    - If we have a nested loop where the inner loop has a separate accumulator that is assigned inside the outer loop (e.g. course_averages), move the accumulator and inner loop into a new function, and call that function from within the original outer loop

5.1 Variable Reassignment and Object Mutation

Variable Reassignment

- *Assignment statement* – __ = __
    - Takes a variable name on the left side and an expression on the right side, and assigns the value of the expression to the variable
- *Variable reassignment* – assigns a value to a variable that already refers to a value

```
>>> x = 1
>>> x = 5  # The variable x is reassigned on this line
```

- A variable reassignment *changes which object a variable refers to*
    - x changes from referring to an object representing 1 to an object representing 5
- Used to update the *accumulator variable* inside the loop

Reassignment is Independent of Prior Uses

- *Variable reassignment only changes the immediate variable being reassigned, and does not change any other variables or objects, even ones that were defined using the variable being reassigned*

```
>>> x = 1
>>> y = x + 2
>>> x = 7
>>> y
3
```

Object Mutation

- *Object mutation* – an operation that changes the value of an existing object
    - Python's list data type contains several methods that *mutate* the given list object rather than create a new one
- Function squares without mutation

```
def squares(nums: List[int]) -> List[int]:
    """Return a list of the squares of the given numbers."""
    squares_so_far = []

    for num in nums:
        squares_so_far = squares_so_far + [num * num]
    return squares_so_far
```

- Function squares with mutation (using list.append)

```
def squares(nums: List[int]) -> List[int]:
    """..."""
    squares_so_far = []

    for num in nums:
        list.append(squares_so_far, num * num)
    return squares_so_far
```

    - squares run by assigning squares_so_far to a single list object before the loop, and then mutating that list object at each loop iteration
    - This code is more efficient that the one without mutation
        - The old one creates a bunch of new list objects
    - squares_so_far is *not* reassigned; instead, the object that it refers to gets mutated
- list.append returns None and has a *side effect* of mutating its list argument

Mutable and Immutable Data Types

- A Python data type is *mutable* when it supports at least one kind of mutating operation
    - It is *immutable* if it does not
- Mutable data types: sets, lists, dictionaries

- Immutable data types: int, float, bool, str
- If we have an object representing the number 3 in Python, that object's value will *always* be 3
    - The variable that refers to this object might be reassigned to a different object later

List vs. *Tuple*, and What's In a *Set*

- A list is *mutable*; a tuple is *immutable*
    - i.e. we can modify a list value by adding an element with list.append; but there is no equivalent tuple.append, nor any other mutating method on tuples
- sets may only contain *immutable* objects; dicts may only contain *immutable keys*
    - We cannot have a set of sets or set of lists in Python
        - Yes it does not match what we have in mathematics
    - But we can have a list of lists

Reasoning About Code with Changing Values

- Reassignment will change which object a variable refers to, sometimes creating a brand new object
- Object mutation changes the object itself, independent of what variable(s) refer to that object
- *Introducing reassignment and mutation makes our code harder to reason about, as we need to track all changes to variable values line by line*
- Avoid using variable reassignment and object mutation when possible
    - Use them in structured code patterns (i.e. the loop accumulator pattern)

5.2 Operations on Mutable Data Types

list.append, list.insert, *and* list.extend

- list.insert – takes a list, an *index*, and an object, and inserts the object at the given index into the list at the given index

```
>>> strings = ['a', 'b', 'c', 'd']
>>> list.insert(strings, 2, 'hello')  # Insert 'hello' into strings at index 2
>>> strings
['a', 'b', 'hello', 'c', 'd']
```

- list.extend – takes two lists and add all items from the second list at the end of the first list

```
>>> strings = ['a', 'b', 'c', 'd']
>>> list.extend(strings, ['CSC110', 'CSC111'])
```

```
>>> strings
['a', 'b', 'c', 'd', 'CSC110', 'CSC111']
```

Assigning to a Specific List Index

- We can overwrite the element stored at a specific index

```
>>> strings = ['a', 'b', 'c', 'd']
>>> strings[2] = 'Hello'
>>> strings
['a', 'b', 'Hello', 'd']
```

- Unlike list.insert, assigning to an index removes the element previously stored at the index from the list

Mutating *sets*

- Two main mutating methods:
  - set.add
  - set.remove
- Re-implement our squares function with set instead of list

```
def squares(numbers: Set[int]) -> Set[int]:
    """..."""
    squares_so_far = set()
    for n in numbers:
        set.add(squares_so_far, n * n)


    return squares_so_far
```

  - set.add will only add the element if the set does not already contain it
  - Sets are unordered whereas list.append will add the element to the end of the sequence

Mutating Dictionaries

- To mutate a dictionary,
  - Add a new key-value pair

```
>>> items = {'a': 1, 'b': 2}
>>> items['c'] = 3
>>> items
{'a': 1, 'b': 2, 'c': 3}
```

    - The left side of the assignment is not a variable but instead an expression representing a component of items (i.e. the key 'c' in the dictionary)

        - When this assignment statement is evaluated, the right side value 3 is stored in the dictionary items as the corresponding value for 'c'
    - Change the associated value for a key-value pair

      >>> items['a'] = 100
      >>> items
      {'a': 100, 'b': 2, 'c': 3}
        - The assignment statement takes an existing key-value pair and replaces the value with a different one

Mutating Data Classes

- Python data classes are mutable by default

  @dataclass
  class Person:
      """A person with some basic demographic information

      Representation Invariants:
      - self.age >= 0

      """
      given_name: str
      family_name: str
      age: int
      address: str
- We mutate instances of data classes by modifying their attributes
    - We do this by assigning to their attributes directly, using *dot notation* on the left side of an assignment statement

      >>> p = Person('David', 'Liu', 100, '40 St. George Street')
      >>> p.age = 200
      >>> p
      Person(given_name='David', family_name='Liu', age=200, address='40 St. George Street')
- Respect the representation invariants when mutating data class instances

5.3 The Python Memory Model: Introduction

Representing Objects

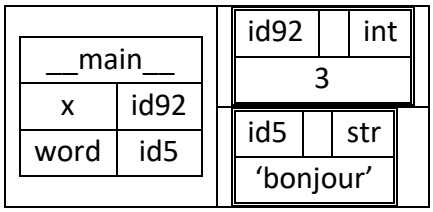
- Every piece of data is stored in a Python program in an *object*

- We cannot control which memory addresses are used to store objects, but we can access a representation of this memory address using the built-in id function

  ```
  >>> id(3)
  1635361280
  >> id('words')
  4297547872
  ```

- *Id* – a unique int identifier to refer to the object
- Every object in Python has three important properties:
  - *Id*
    - The only one among the three guaranteed to be unique
  - *Value*
  - *Type*
- A variable is not an object and so does not actually store data
  - Variables store an id hat *refers* to an object that stores data
    - i.e. variables *contain* the id of an object
- With a full object-based Python memory model, we draw one table-like structure on the left showing the mapping between variables and object ids, and the the objects on the right
  - Each object is represented as a box, with its id in the upper-left corner, type in the upper-right corner, and value in the middle
  - The actual object id reported by the id function is unimportant
    - We just need to know that each object has a unique identifier

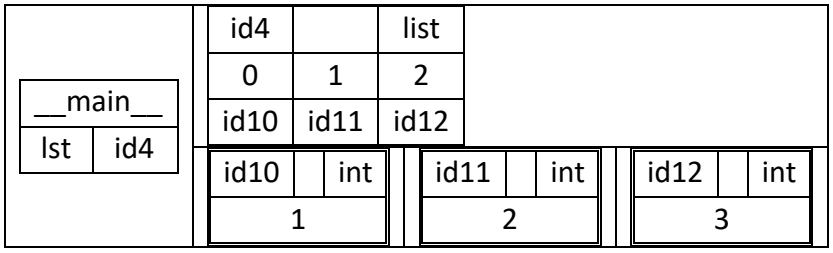
  ```
  >>> x = 3
  >>> word = 'bonjour'
  ```



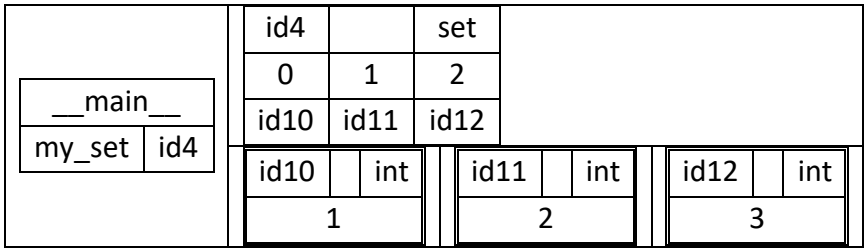
  - There is no 3 inside the box for variable x; instead, there is the *id* of an object whose value is 3
- Assignment statements and evaluating expressions
  - Evaluating an expression
    - Produces and id of an object representing the value of the expression
  - Assignment statements
    - 1. Evaluate the expression on the right side, yielding the id of an object
    - 2. If the variable on the left side doesn't already exist, create it
    - 3. Store the id from the expression on the right side in the variable on the left side

Representing Compound Data

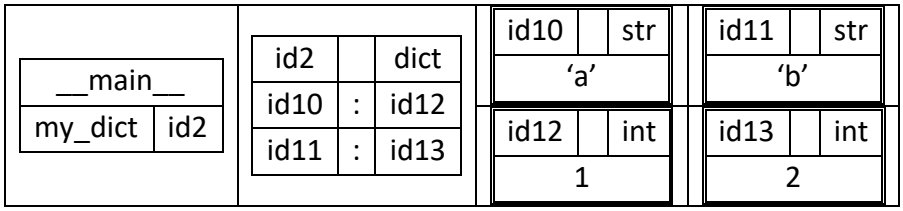
- An instance of a compound data type does not store values directly
    - Instead, it stores the ids of other objects
- Lists
    - lst = [1, 2, 3]



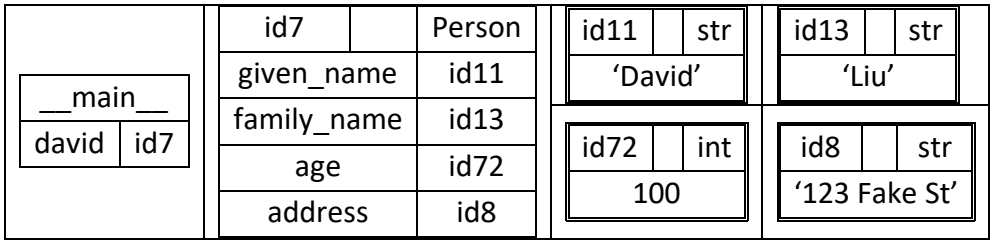
        - 4 separate objects on the diagram:
            - One for each of the ints 1, 2, 3
            - One for the list
- Sets
    - my_set = {1, 2, 3}



- Dictionaries
    - my_dict = {'a': 1, 'b': 2}



        - 5 objects in total
- Data classes
    - For the Person object



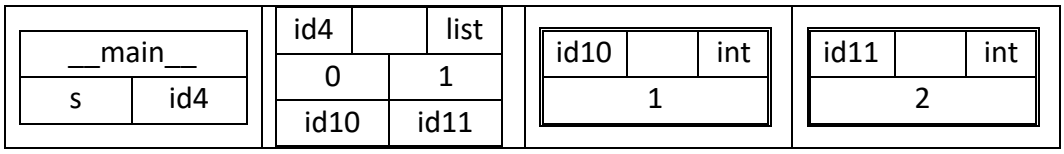
- Use the convention of drawing a *double box* around objects that are immutable

Visualizing Variable Reassignment and Object Mutation

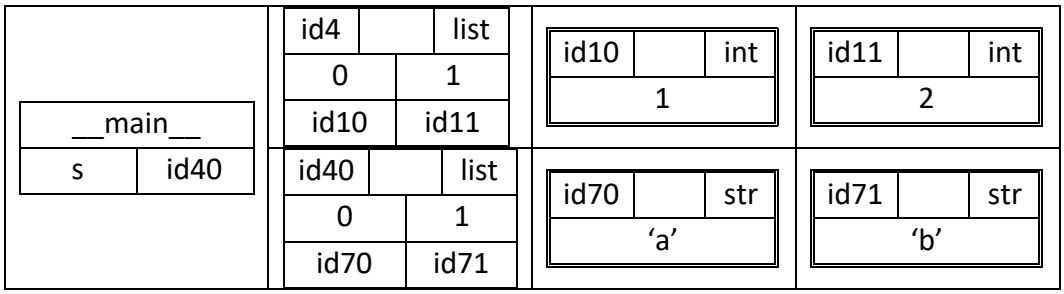
- Consider this case of variable reassignment

```
>>> s = [1, 2]
>>> s = ['a', b']
```

  - Before reassignment



  - After reassignment

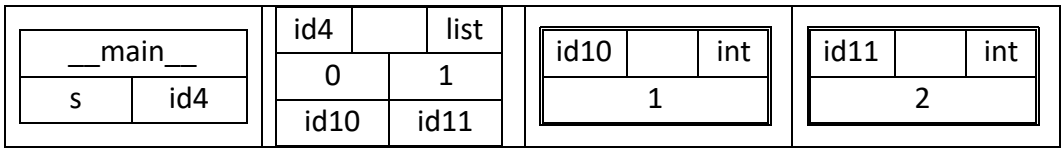


  - The original list object [1, 2] is not mutated
    - Variable reassignment *does not mutate any objects*
    - What the variable refers to is changed

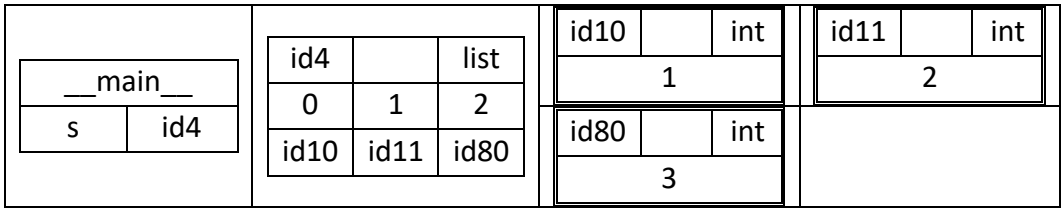
- Consider this case of object mutation

```
>>> s = [1, 2]
>>> list.append(s, 3)
```

  - Before mutation



  - After mutation

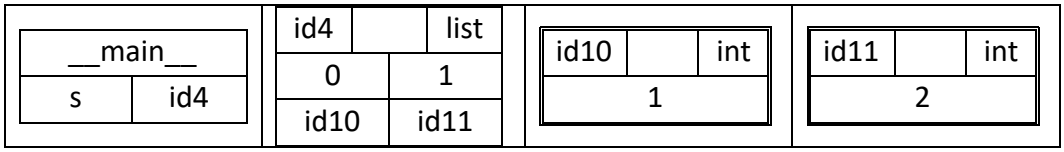


  - No new list object is created
    - The list object [1, 2] is mutated, and a third id is added at its end
  - The id of s is not changed despite changing in size

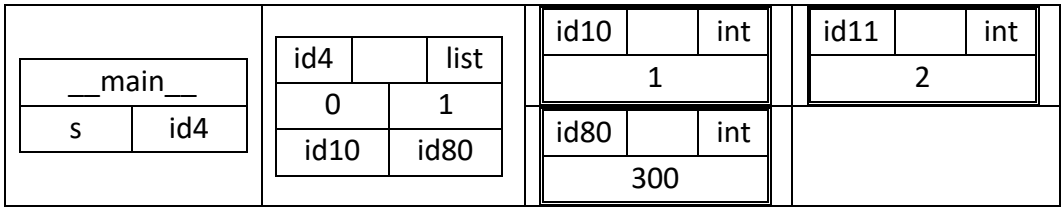
- Consider this case of assigning to part of a compound data type

```
>>> s = [1, 2]
>>> s[1] = 300
```

- Before mutation



- After mutation



- Rather than reassigning a variable, it reassigns an id that is part of an object
- This statement *does* mutate an object, and doesn't reassign any variables

5.4 Aliasing and "Mutation at a Distance"

Aliasing

- Let v1 and v2 be Python variables. v1 and v2 are *aliases* when they refer to the same object
- Consider the following

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x  # make z refer to the object that x refers to
```

  - x and z are aliases, as they both reference the same object
    - They have the same id
  - x and y are two different list objects stored separately in the computer's memory

Aliasing and Mutation

- Aliasing allows "mutation at a distance"

```
>>> x = [1, 2, 3]
>>> z = x
>>> z[0] = -999
>>> x
[-999, 2, 3]
```

  - The third line mutates the object that z refers to
- Another example

```
>>> x = [1, 2, 3]
>>> z = x
>>> y[0] = -999
```

```
>>> x
[1, 2, 3]
```

Variable Reassignment

- Example of variable reassignment

```
>>> x = (1, 2, 3)
>>> z = x
>>> z = (1, 2, 3, 40)
>>> x
(1, 2, 3)
```

  - When we change z on the third line, x does *not* change this time
  - We reassigned z to a new object, which has no effect on the object that x refers to
- Reassigning *breaks the aliasing*
  - Afterwards the two variables would no longer refer to the same object

Aliasing and Loop Variables

- In element-based for loops, the loop variable is an alias to one of the objects *within* the collection
- Example

```
>>> numbers = [5, 6, 7]
>>> for number in numbers:
…       number = number + 1
…
>>> numbers
[5, 6, 7]
```

  - The assignment statement inside the for loop simply changes what the loop variable refers to, but does not change what the contents of the list numbers refers to
- Assignment statements of the form <name> = _____ *reassign* the variable <name> to a new value
- Assignment statements of the form <name>[<index>] = _____ *mutate* the list object that <name> currently refers to

Two Types of Equality

- *Value equality* – whether the two objects have the same *value*
  - x == y
- *Identity equality* – whether two objects have the same *ids*

    - x is y
- Identity equality is a stronger property than value equality
    - For all objects a and b, if a is b then a == b
        - The converse is not true

Aliasing with Immutable Data Types

- Aliasing exists for immutable data types, but there is never any "action at a distance"
    - Because immutable values can never change
- Automatic aliasing of (some) immutable objects
    - If two variables have the same immutable value, the program's behaviour does not depend on whether the two variables are aliases or not
    - The Python interpreter saves some computer memory by not creating new objects for some immutable values
        - i.e. every occurrence of the boolean value True refers to the same object
        - "Small" integers are automatically aliased, while "large" integers are not
    - The Python interpreter takes the object creation "shortcut" with *some* string values
- Rules for object comparisons:
    - For *boolean* values, use is to compare for equality
    - For *non-boolean immutable* values, use == to compare for equality
        - Using is can lead to surprising results
    - For *mutable* values, use == to compare value equality
    - For *mutable* values, use is to check for aliasing

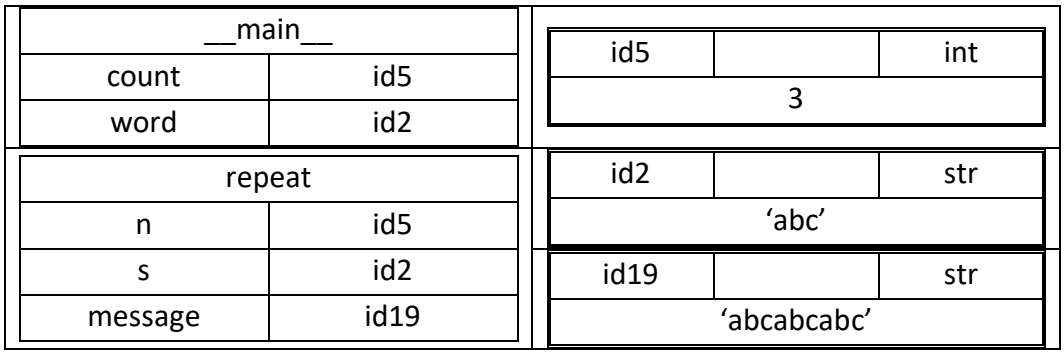
5.5 The Full Python Memory Model: Function Calls

Stack Frames

- Suppose we define the following function, and then call it in the Python console

```
def repeat(n: int, s:str) -> str:
    message = s * n
    return message

# In the Python console
>>> count = 3
>>> word = 'abc'
>>> result = repeat(count, word)
```

- When repeat(count, word) is called, immediately before the return message statement executes, the full Python memory model diagram would look like the following



  - The variables are separated into 2 separate boxes
    - One for the Python console and one for the function call for repeat
  - All variables, regardless of which box they're in, store only ids that refer to objects on the right side
  - count and n are aliases
  - word and s are aliases

- *Stack frame* – the boxes on the left side of the diagram
  - Also known as *frame*
  - Keep track of the functions that have been called in a program, and the variables defined within each function
- *Function call stack* – the collection of stack frames
- Every time we call a function, the Python interpreter does the following:
  - 1. Create a new stack frame and add it to the call stack
  - 2. Evaluate the arguments in the function call, yielding the ids of objects (one per argument).
    - Each of these ids is assigned to the corresponding parameter, as an entry in the new stack frame
  - 3. Execute the body of the function
  - 4. When a return statement is executed in the function body, the id of the returned object is saved and the stack frame for the function call is removed from the call stack

Argument Passing and Aliasing

- When we called repeat(count, word), it is as if we wrote

  n = count

  s = word

before executing the body of the function

- This aliasing allows us to define functions that mutate their argument values
- Example

```
def emphasize(words: List[str]) -> None:
    """Add emphasis to the end of a list of words."""
    new_words = ['believe', 'me!']
    list.extend(words, new_words)

# In the Python console
>>> sentence = ['winter', 'is', 'coming']
>>> emphasize(sentence)
>>> sentence
['winter', 'is', 'coming', 'believe', 'me!']
```

  - words and sentence are aliases, and so mutating words within the function causes a change to occur in __main__ as well

- Another example

```
def emphasize_v2(words: List[str]) -> None:
    """..."""
    new_words = ['believe', 'me!']
    words = words + new_words

# In the Python console
>>> sentence = ['winter', 'is', 'coming']
>>> emphasize_v2(sentence)
>>> sentence
['winter', 'is', 'coming']
```

  - List concatenation with + creates a new list object
  - words and sentence are no longer alias after words is reassigned

5.6 Testing Functions III: Testing Mutation

Intro

- If a function's documentation does not specify that an object will be mutated, then it *must not* be mutated

Identifying Mutable Parameters

- Consider the squares function

```
def squares(nums: List[int]) -> List[int]:
```

```
    """Return a list of the squares of the given numbers."""
    squares_so_far = []

    for num in nums:
        list.append(squares_so_far, num * num)
    return squares_so_far
```

- Because squares_so_far is created by the function squares, it is okay that it is mutated (i.e. the call to list.append inside the loop)
- The nums list is passed as an argument to squares
  - Because the docstring does not indicate that nums will be mutated, it is expected that the squares function will not mutate the list object referred to by nums

- Contrast the above program with the following where the function *does* mutate its input

```
def square_all(nums: List[int]) -> None:
    """Modify nums by squaring each of its elements."""
    for i in range(0, len(nums)):
        nums[i] = nums[i] * nums[i]
```

Testing for No Mutation

- Write a test that ensures the squares function does not mutate the list referred to by nums

```
def test_squares_no_mutation() -> None:
    """Test that squares does not mutate the list it is given."""
    lst = [1, 2, 3]
    squares(lst)

    assert lst == [1, 2, 3]
```

- In order to test that a list is not mutated,
  - We first create a list lst
  - We then call the squares function on lst
    - We do not assign the result to a variable because we don't care about the returned value for the purpose of this test
  - We add an assertion that ensures lst has not been mutated
- Our assertion checks that *after* the call to squares, lst still has value [1, 2, 3]
- Another way to accomplish this, without re-typing the list value, is by creating a copy of lst before the call to squares

```
def test_squares_no_mutation() -> None:
    """Test that squares does not mutate the list it is given."""
```

```
    lst = [1, 2, 3]
    lst_copy = list.copy(lst)  # Create a copy of lst (not an alias)
    squares(lst)

    assert lst == lst_copy
```

- We create the list and its copy before the call to squares
- We test for mutation (i.e. the assertion) after the call to squares
- Generalizing this test
    - For the above test, if we replaced lst's value with another list, the test would behave in the exact same way
    - Our test is suitable to be generalized into a *property-based test*, representing the following property:
        - For all lists of integer lst, calling squares(lst) does not mutate lst

```
from typing import List

from hypothesis import given
from hypothesis.strategies import lists, integers

@given(lst=lists(integers()))
def test_squares_no_mutation_general(lst: List[int]) -> None:
    """Test that squares does not mutate the list it is given."""
    lst_copy = list.copy(lst)  # Create a copy of lst (not an alias)
    squares(lst)

    assert lst == lst_copy
```

Testing for Mutation

- Consider testing the square_all function

```
def test_square_all() -> None:
    """Test that square_all mutates the list it is given correctly."""
    lst = [1, 2, 3]
    result = squares_all(lst)

    assert result == [1, 4, 9]
```

    - This test fails because square_all returns None
        - None == [1, 4, 9] is False
- We test if the value of lst has changed

```
def test_square_all_mutation() -> None:
```

```
    """Test that square_all mutates the list it is given correctly. """
    lst = [1, 2, 3]
    square_all(lst)

    assert lst == [1, 4, 9]
```

- The above test can be generalized into a property-based test by storing a copy of the original list and verifying the relationship between corresponding elements

```
@given(lst=lists(integers()))
def test_square_all_mutation_general(lst: List[int]) -> None:
    """Test that square_all mutates the list it is given correctly."""
    lst_copy = list.copy(lst)
    square_all(lst)

    assert all({lst[i] == lst_copy[i] ** 2 for i in range(0, len(lst))})
```

6.1 An Introduction to Number Theory

Divisibility, Primality, and the Greatest Common Divisor

- $Definition.$ Let $n, d \in \mathbb{Z}$. We say that d **divides** n, when there exists a $k \in \mathbb{Z}$ such that $n = dk$. We use the notation $d \mid n$ to represent the statement "d divides n"
  - The following phrases are synonymous with "d divides n":
    - n **is divisible by** d
    - d is a **factor** of n
    - n is a **multiple** of d
- $Definition.$ Let $p \in \mathbb{N}$. We say p is **prime** when it is greater than 1 and the only natural numbers that divide it are 1 and itself
- $Definition.$ Let $x, y, d \in \mathbb{Z}$. We say that d is a **common divisor** of x and y when d divides x and d divides y
- We say that d is the **greatest common divisor** of x and y when it is the largest number that is a common divisor of x and y, or – when x and y are both 0.
  - We can define the function $\gcd : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$ as the function which takes numbers x and y, and returns their greatest common divisor
- 1 divides every natural number, so 1 is a common divisor between any two natural numbers
- $Definition.$ Let $m, n \in \mathbb{Z}$. We say that m and n are **coprime** when $\gcd(m, n) = 1$

Quotients and Remainders

- For all $n \in \mathbb{Z}$ and $d \in \mathbb{Z}^+$, there exist $q \in \mathbb{Z}$ and $r \in \mathbb{N}$ such that $n = qd + r$ and $0 \leq r < d$. Moreover, these q and r are *unique* for a given n and d
    - We say that q is the **quotient** when n is divided by d, and that r is the **remainder** when n is divided by d
- In Python, for given integers n and d, we can compute their quotient using //, their remainder using %, and both at the same time using the built-in function divmod

```
>>> 9 // 2
4
>>> 9 % 2
1
>>> divmod(9, 2)
(4, 1)
```

Modular Arithmetic

- We often care about the *remainder* when we divide a number by another
- $Definition.$ Let $a, b, n \in \mathbb{Z}$ with $n \neq 0$. We say that a is **equivalent to** b **modulo** n when $n \mid a - b$. In this case, we write $a \equiv b \pmod{n}$
- Modular equivalence can be used to divide up numbers based on their remainders when divided by n
    - Let $a, b, n \in \mathbb{Z}$ with $n \neq 0$. Then $a \equiv b \pmod{n}$ if and only if a and b have the same remainder when divided by n
- Let $a, b, c, n \in \mathbb{Z}$ with $n \neq 0$. Then the following hold:
    - $a \equiv a \pmod{n}$
    - If $a \equiv b \pmod{n}$ then $b \equiv a \pmod{n}$
    - If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ the $a \equiv c \pmod{n}$
- Let $a, b, c, d, n \in \mathbb{Z}$ with $n \neq 0$. If $a \equiv c \pmod{n}$ and $b \equiv d \pmod{n}$, then the following hold:
    - $a + b \equiv c + d \pmod{n}$
    - $a - b \equiv c - d \pmod{n}$
    - $ab \equiv cd \pmod{n}$
- Addition, subtraction, and multiplication operations preserve modular equivalence relationships
    - However, this is *not* the case with division

6.2 Proofs with Number Theory

Intro

- *Mathematical proof* – how we communicate ideas about the truth or falsehood of a statement to others
- A proof is made of *communication*, from the person creating the proof to the person digesting it
- Audience of our proof: an average computer science student
    - Formal
    - No assuming much background knowledge

First Examples

- Four parts that leads to a completed proof:
    - 1. The statement that we want to prove or disprove
    - 2. A translation of the statement into predicate logic
        - Provides insight into the *logical structure* of the statement
    - 3. A discussion to try to gain some intuition about why the statement is true
        - Informal
        - Usually reveals the mathematical insight that forms the content of a proof
        - The hardest part of developing a proof
    - 4. A formal proof
        - The "final product" of our earlier work
- **Ex**. Prove that 23 | 115
    - $Translation.$ We will *expand* the definition of divisibility to rewrite this statement in terms of simpler operations:
    $$\exists \mathrm{k} \in \mathrm{Z}, 115 = 23\mathrm{k}$$
    - $Discussion.$ We just need to divide 115 by 23
    - $Proof.$ let $k = 5$
    - Then $115 = 23 \cdot 5 = 23 \cdot k$ QED
- A typical proof of an existential
    - Given statement to prove: $\exists x \in S, P(x)$
    - $Proof.$ Let $x =$ _____
    - [Proof that P(____) is True.] QED
    - The two blanks represent the same element of S, which *we* get to choose as a prover
- **Ex**. Prove that there exists an integer that divides 104
    - $Translation.$ We could write $\exists a \in \mathbb{Z}, a \mid 104$. Expanding the definition of divisibility,
    $$\exists a, k \in \mathbb{Z}, 104 = ak$$
    - $Discussion.$ We get to pick both a and k. Any pair of divisors will work

    - $Proof$. Let $a = -2$ and let $k = -52$
    - Then $104 = ak$ QED
- A *mathematical proof* must introduce all variables contained in the sentence being proven

Alternating Quantifiers

- **Ex**. Prove that all integers are divisible by 1
    - $Translation$. The statement contains a universal quantification: $\forall n \in \mathbb{Z}, 1 \mid n$. Unpacking the definition of divisibility,
    $$\forall n \in \mathbb{Z}, \exists\, k \in \mathbb{Z}, n = 1 \cdot k$$
    - $Discussion$. The statement is valid when k equals n. Introduce the variables in the same order they are quantified in the statement
    - $Proof$. Let $n \in \mathbb{Z}$. Let $k = n$
    - Then $n = 1 \cdot n = 1 \cdot k$ QED
- A typical proof of a universal
    - Given statement to prove: $\forall x \in S, P(x)$
    - $Proof$. Let $x \in S$. (i.e. let x be an arbitrary element of S)
    - [Proof that $P(x)$ is True]. QED
- Any existentially-quantified variable can be assigned a value that depends on the variables defined before it
    - In programming, we first initialize a variable n, and then define a new variable k that is assigned the value of n
- The order of variables in the statement determines the order in which the variables must be introduced in the proof
    - And hence which variables can depend on which other variables
- We cannot use a variable before it's defined

Proofs Involving Implications

- **Ex**. Prove that for all integers x, if x divides $(x + 5)$, then x also divides 5
    - $Translation$. There is both a universal quantification and implication in this statement
    $$\forall x \in \mathbb{Z}, x \mid (x + 5) \Rightarrow x \mid 5$$
    - Unpacking the definition of divisibility,
    $$\forall x \in \mathbb{Z}, (\exists k_1 \in \mathbb{Z}, x + 5 = k_1 x) \Rightarrow (\exists k_2 \in \mathbb{Z}, 5 = k_2 x)$$
    - $Discussion$. We are going to *assume* that x divides $x + 5$, and we need to *prove* that x divides 5
    - Since x id divisible by x, we should be able to subtract it from $x + 5$ and keep the result a multiple of x

- We need to "turn" the equation $x + 5 = k_1 x$ into the equation $5 = k_2 x$
- $Proof$. Let x be an arbitrary integer. *Assume* that $x \mid (x + 5)$, i.e., that there exists $k_1 \in \mathbb{Z}$ such that $x + 5 = k_1 x$. We want to prove that there exists $k_2 \in \mathbb{Z}$ such that $5 = k_2 x$
- Let $k_2 = k_1 - 1$

$$\begin{aligned} k_2 x &= (k_1 - 1)x \\ &= k_1 x - x \\ &= (x + 5) - x \text{ (we assumed } x + 5 = k_1 x) \\ &= 5 \end{aligned}$$

- To prove an implication, we need to assume that the hypothesis is True, and then prove that the conclusion is also True
- The previous example is *not* a claim that $x \mid (x + 5)$ is True; rather, it is a way to consider what would happen *if* $x \mid (x + 5)$ is True
- A typical proof of an implication (direct)
  - Given statement to prove: $p \Rightarrow q$
  - $Proof$. Assume p
  - [Proof that q is True.] QED

<u>Generalizing our Example</u>

- *Generalization* – taking a true statement (and a proof of the statement) and then replacing a concrete value in the statement with a universally quantified variable
- Ex. Prove that for all $d \in \mathbb{Z}$, and for all $x \in \mathbb{Z}$, if x divides $(x + d)$, then x also divides d
  - $Translation$. Same translation as the previous example, except now we have an extra variable:

$$\forall d, x \in \mathbb{Z}, (\exists k_1 \in \mathbb{Z}, x + d = k_1 x) \Rightarrow (\exists k_2 \in \mathbb{Z}, d = k_2 x)$$

  - $Discussion$. We should be able to use the same set of calculations as the previous example
  - $Proof$. Let d and x be arbitrary integers. *Assume* that $x \mid (x + d)$, i.e., there exists $k_1 \in \mathbb{Z}$ such that $x + d = k_1 x$. We want to prove that there exists $k_2 \in \mathbb{Z}$ such that $d = k_2 x$.
  - Let $k_2 = k_1 - 1$

$$\begin{aligned} k_2 x &= (k_1 - 1)x \\ &= k_1 x - x \\ &= (x + d) - x \\ &= d \end{aligned}$$

    QED
  - The above proof is basically the same as the previous one
    - We have simply swapped out all the 5's with d's

        - The proof *did not depend on the value* 5
        - The original statement and proof *generalize* to this second version
- The more general the statement, the more useful it becomes
    - i.e. the exponent laws
        - They apply to every possible number, regardless of what our concrete calculation is
        - We know we can use them in our calculations

6.3 Proofs and Algorithms I: Primality Testing

Intro

- Function that determines whether p is prime

```
def is_prime(p: int) -> bool:
    """Return whether p is prime."""
    possible_divisors = range(1, p+1)
    return (
        p > 1 and
        all({d == 1 or d == p for d in possible_divisors if divides(d, p)})
    )
```

    - It is a direct translation of the mathematical definition of prime numbers, with the only difference being our restriction of the range of possible divisors
    - This algorithm is inefficient because it checks more numbers than necessary
    - The range of possible divisors extends only to the square root of the input p

```
from math import floor, sqrt

def is_prime(p: int) -> bool:
    """Return whether p is prime."""
    possible_divisors = range(2, floor(sqrt(p)) + 1)
    return (
        p > 1 and
        all({not divides(d, p) for d in possible_divisors})
    )
```

    - This version is faster as the range of possible divisors to check is smaller
    - We need proofs to know that this version of is_prime is correct

A Property of Prime Numbers

- **Theorem.** Let $p \in \mathbb{Z}, Prime(p) \Leftrightarrow (p > 1 \wedge (\forall d \in \mathbb{N}, 2 \leq d \leq \sqrt{p} \implies d \nmid p))$
- $p \Leftrightarrow q$ is equivalent to $(p \implies q) \wedge (q \implies p)$

    - To argue that a biconditional is True, we do so by proving the two different implications
- A typical proof of a biconditional
    - Given statement to prove: $p \Leftrightarrow q$
    - $Proof$. This proof is divided into two parts.
    - Part 1 $(p \Longrightarrow q)$: Assume p
    - [Proof that q is True]
    - Part 2 $(q \Longrightarrow p)$: Assume q
    - [Proof that p is True]
- Proving the first implication
    - $Discussion$. We assume that p is prime, and will need to prove:
        - $p > 1$
        - $\forall d \in \mathbb{N}, 2 \leq d \leq \sqrt{p} \Longrightarrow d \nmid p$
    - Definition of prime in predicate logic:

      $Prime(p) : p > 1 \wedge (\forall d \in \mathbb{N}, d \mid p \Longrightarrow d = 1 \vee d = p)$
    - If d is between 2 and $\sqrt{p}$, then it can't equal 1 or p, which are the only possible divisors of p
    - $Proof$. Let $p \in \mathbb{Z}$ and assume that p is prime. We need to prove that $p > 1$ and for all $d \in \mathbb{N}$, if $2 \leq d \leq \sqrt{p}$ then d does not divide p
    - **Part 1**: proving that $p > 1$
    - By the definition of prime, we know that $p > 1$
    - **Part 2:** proving that for all $d \in \mathbb{N}$, if $2 \leq d \leq \sqrt{p}$ then d does not divide p
    - First, since $2 \leq d$, we know $d > 1$, and so $d \neq 1$. Second, since $p > 1$, we know that $\sqrt{p} < p$, and so $d \leq \sqrt{p} < p$
    - This means that $d \neq 1$ and $d \neq p$. By the definition of prime, we can conclude that $d \nmid p$ QED
    - When the input p is a prime number, we know that the expressions p > 1 and all(not divides(d, p) for d in possible_divisors) will both evaluate to True, and so the function will return True
        - We've proven that is_prime returns the correct value for *every* prime number, without a single test case
- Proving the second implication
    - At this point, we've said nothing about how is_prime behaves when given a non-prime number
    - $Discussion$. We now need to prove the converse of the first implication: if $p > 1$ and $\forall d \in \mathbb{N}, 2 \leq d \leq \sqrt{p} \Longrightarrow d \nmid p$, then p must be prime

- Expanding the definition of prime, we need to prove that $p > 1$ (which we've assumed) and for all $d_1 \in \mathbb{N}, d_1 \mid p \implies d_1 = 1 \vee d_1 = p$
- Idea: Let $d_1 \in \mathbb{N}$ and assume $d_1 \mid p$, and use the condition that $\forall d \in \mathbb{N}, 2 \leq d \leq \sqrt{p} \implies d \nmid p$ to prove that d_1 is 1 or p
- $Proof$. Let $p \in \mathbb{N}$, and assume $p > 1$ and that $\forall d \in \mathbb{N}, 2 \leq d \leq \sqrt{p} \implies d \nmid p$. We want to prove that p is prime, i.e., that $p > 1$ and that $d_1 \in \mathbb{N}, d_1 \mid p \implies d_1 = 1 \vee d_1 = p$
- For the first part, $p > 1$ is true because it's one of our assumptions. For the second part, first let $d_1 \in \mathbb{N}$, and assume $d_1 \mid p$. We'll prove that $d_1 = 1 \vee d_1 = p$
- From our second assumption, we know that since $d_1 \mid p$, it is not between 2 and $\sqrt{p}$. So then either $d_1 < 2$ or $d_1 > \sqrt{p}$. We divide our proof into two *cases* based on these possibilities
- **Case 1:** assume $d_1 < 2$
- Since $d_1 \in \mathbb{N}$, it must be 0 or 1 in this case. We know $0 \nmid p$ because $p > 1$, and so $d_1 = 1$
- **Case 2:** assume $d_1 > \sqrt{p}$
- Since we assumed $d_1 \mid p$, we expand the definition of divisibility to conclude that $\exists k \in \mathbb{Z}, p = d_1 k$. Since $d_1 > \sqrt{p}$ in this case, we know that $k = \frac{p}{d_1} < \frac{p}{\sqrt{p}} = \sqrt{p}$
- Since $p = d_1 k$, we know that $k \mid p$ as well, and so our second assumption applied to k tells us that k is not between 2 and $\sqrt{p}$
- So $k < \sqrt{p}$ and is not between 2 and $\sqrt{p}$. Therefore $k = 1$, and so $d_1 = \frac{p}{k} = p$
- QED
- What we've proved is that if is_prime(p) returns True, then p must be prime.
  - The contrapositive: if p is *not* prime, then is_prime(p) returns False

- Putting the two implications together, we have:
  - For all integers p, if p is prime then is_prime(p) returns True
  - For all integers p, if is_prime(p) returns True the p is prime
- Since every integer p is either prime or not prime, we can conclude that this implementation of is_prime is *correct* according to its specification

Algorithm Correctness and Theoretical Properties

- The correctness of our *algorithm* is derived from the *theoretical properties of prime numbers* that we expressed in formal predicate logic

6.4 Proof by Cases and Disproofs

Proof by Cases

- When different arguments are required for different elements, we divide the domain into different parts, and then write a separate argument for each part
- We pick a set of unary predicates $P_1, P_2, \dots, P_k$ (for some positive integer k), such that for every element x in the domain, x satisfies at least one of the predicates (we say that these predicates are *exhaustive*)
- In our previous example, we started with a domain "$d_1 \in \mathbb{N}$" and then narrowed this to "$d_1 \in \mathbb{N}\ and\ (d_1 < 2 \vee d_1 > \sqrt{p})$", leading to the following predicates for our cases:

$$P_1(d_1) : d_1 < 2, \qquad P_2(d_1) : d_1 > \sqrt{p}$$

- Then, we divide the proof body into cases, where in each case we *assume* that one of the predicates is True, and use that assumption to construct a proof that specifically works under that assumption
- A typical proof by cases
    - Given statement to prove: $\forall x \in S, P(x)$. Pick a set of exhaustive predicates $P_1, \dots, P_k$ of S
    - $Proof$. Let $x \in S$. We will use proof by cases
    - **Case 1.** *Assume* $P_1(x)$ is True
    - [Proof that $P(x)$ is True, assuming $P_1(x)$]
    - **Case 2.** *Assume* $P_2(x)$ is True
    - [Proof that $P(x)$ is True, assuming $P_2(x)$]
    - ⋮
    - **Case k.** *Assume* $P_k(x)$ is True
    - [Proof that $P(x)$ is True, assuming $P_k(x)$] QED
- A *simple* proof which works for all elements of the domain is preferable than a proof by cases

Cases and the Quotient-Remainder Theorem

- **Theorem.** (Quotient-Remainder Theorem) For all $n \in \mathbb{Z}$ and $d \in \mathbb{Z}^+$, there exist $q \in \mathbb{Z}$ and $r \in \mathbb{N}$ such that $n = qd + r$ and $0 \leq r < d$. Moreover, these q and r are *unique* for a given n and d
- We say that q is the *quotient* when n is divided by d, and that r is the *remainder* when n is divided by d
- This theorem tells us that for any non-zero divisor $d \in \mathbb{Z}^+$, we can separate all possible integers into d different groups, corresponding to their possible remainders (between 0 and $d - 1$) when divided by d
- **Ex**. Prove that for all integers x, $2 \mid x^2 + 3x$
    - $Translation$. Expanding the definition of divisibility,

$$\forall x \in \mathbb{Z}, \exists k \in \mathbb{Z}, x^2 + 3x = 2k$$

- $Discussion$. We want to "factor out a 2" from the expression $x^2 + 3x$, but this only works if x is even. If x is odd, then both x^2 and $3x$ will be odd, and adding 2 odd numbers together produces an even number.
- With the Quotient-Remainder Theorem, we can:
  - "Know" that every number has to be either even or odd
  - Formalize the algebraic operations of "factoring out a 2" and "adding 2 odd numbers together"
- $Proof$. Let $x \in \mathbb{Z}$. By the Quotient-Remainder Theorem, we know that when x is divided by 2, the two possible remainders are 0 and 1.
- **Case 1:** assume the remainder when x is divided by 2 is 0, i.e., we assume there exists $q \in \mathbb{Z}$ such that $x = 2q + 0$. We will show that there exists $k \in \mathbb{Z}$ such that $x^2 + 3x = 2k$
- We have:

$$x^2 + 3x = (2q)^2 + 3(2q)$$
$$= 4q^2 + 6q$$
$$= 2(2q^2 + 3q)$$

- So let $k = 2q^2 + 3q$. Then $x^2 + 3x = 2k$
- **Case 2:** assume the remainder when x is divided by 2 is 1, i.e., we assume there exists $q \in \mathbb{Z}$ such that $x = 2q + 1$. We will show that there exists $k \in \mathbb{Z}$ such that $x^2 + 3x = 2k$
- We have:

$$x^2 + 3x = (2q + 1)^2 + 3(2q + 1)$$
$$= 4q^2 + 4q + 1 + 6q + 3$$
$$= 2(2q^2 + 5q + 2)$$

- So let $k = 2q^2 + 5q + 2$. Then $x^2 + 3x = 2k$ QED

<u>False Statements and Disproofs</u>

- An absence of proof is not enough to convince us that the statement is False
- *Disproof* – a proof that the *negation* of the statement is True
- **Ex.** Disprove the following statement: every natural number divides 360
  - $Translation$. The statement can be written as $\forall n \in \mathbb{N}, n \mid 360$
  - To prove it False, we need its negation

$$\neg(\forall n \in \mathbb{N}, n \mid 360)$$
$$\exists n \in \mathbb{N}, n \nmid 360$$

  - $Discussion$. The number 7 doesn't divide 360
  - $Proof$. Let $n = 7$
  - Then $n \nmid 360, since \frac{360}{7} = 51.428\ldots$ is not an integer. QED

- *Counterexample* – the value that causes the predicate to be False (or causes the negation of the predicate to be True)
    - The value 7 is the *counterexample* for the previous example
- A typical disproof of a universal (counterexample)
    - Given statement to *disprove*: $\forall x \in S, P(x)$
    - $Proof$. We prove the negation, $\exists x \in S, \neg P(x)$. Let x = _____
    - [Proof that $\neg P$(_____) is True] QED

6.5 Greatest Common Divisor

Intro

- $Definition$. Let $x, y, d \in \mathbb{Z}$. We say that d is a **common divisor** of x and y when d divides x and d divides y
- We say that d is the **greatest common divisor** of x and y when it is the largest number that is a common divisor or x and y, or 0 when x and y are both 0
- We can define the function gcd: $\mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$ as the function which takese numbers x and y, and returns their greatest common divisor
- If e is any number which divides m and n, then $e \leq d$
- Let $m, n, d \in \mathbb{Z}$, and suppose $d = \gcd(m, n)$, then d satisfies the following:

$$(m = 0 \wedge n = 0 \Longrightarrow d = 0) \wedge$$
$$(m \neq 0 \vee n \neq 0 \Longrightarrow d \mid m \wedge d \mid n \wedge (\forall e \in \mathbb{N}, e \mid m \wedge e \mid n \Longrightarrow e \leq d))$$

- **Ex.** Prove that for all integers p and q, if p and q are distinct primes, then p and q are *coprime*, meaning $\gcd(p, q) = 1$
    - $Translation$. Structure of the above statement:
    $$\forall p, q \in \mathbb{Z}, (Prime(p) \wedge Prime(q) \wedge p \neq q) \Longrightarrow \gcd(p, q) = 1$$
    - We could unpack the definitions of $Prime$ and gcd, but it would not be necessary
        - To show that $\gcd(p, q) = 1$, we just need to make sure that neither p nor q divides the other
    - $Proof$. Let $p, q \in \mathbb{Z}$. Assume that p and q are both prime, and that $p \neq q$. We want to prove that $\gcd(p, q) = 1$
    - By the definition of prime, $p \neq 1$ (since $p > 1$)
    - The only positive divisors of q are 1 and q itself
    - Since we assumed $p \neq q$ and concluded $p \neq 1$, we know that $p \nmid q$
    - Since we know that 1 divides every number, 1 is the only positive common divisor of p and q, so $\gcd(p, q) = 1$ QED

Linear Combinations and the Greatest Common Divisor

- $Definition.$ Let $m, n, a \in \mathbb{Z}$. We say that a is a **linear combination** of m and n when there exist $p, q \in \mathbb{Z}$ such that $a = pm + qn$
    - i.e. 101 is a linear combination of 5 and 3, since $101 = 10 \cdot 5 + 17 \cdot 3$
- **Theorem.** (*Divisibility of Linear Combinations*) Let $m, n, d \in \mathbb{Z}$. If d divides m and d divides n, then d divides every linear combination of m and n
- **Theorem.** (*GCD Characterization*) Let $m, n, d \in \mathbb{Z}$, and assume at least one of them is non-zero. Then $\gcd(m, n)$ is the smallest positive integer that is a linear combination of m and n
- **Ex.** For all $m, n, d \in \mathbb{Z}$, if d divides both m and n then d also divides $\gcd(m, n)$
    - $Translation.$ We can translate this statement as
    $$\forall m, n, d \in \mathbb{Z}, d \mid m \wedge d \mid n \Longrightarrow d \mid \gcd(m, n)$$
    - $Discussion.$ By the GCD Characterization Theorem, we can write $\gcd(m, n)$ as $pm + qn$
        - Any number that divides m and n will divide $pm + gn$ as well
    - $Proof.$ Let $m, n, d \in \mathbb{Z}$. Assume that $d \mid m$ and $d \mid n$. We want to prove that $d \mid \gcd(m, n)$. We'll divide our proof into two cases.
    - **Case 1:** assume $m = 0$ and $n = 0$
    - By the definition of gcd we know that $\gcd(m, n) = 0$. So $d \mid \gcd(m, n)$, since we assumed that d divides 0
    - **Case 2:** assume $m \neq 0$ or $n \neq 0$
    - By the GCD Characterization Theorem, there exists integers $p, q \in \mathbb{Z}$ such that $\gcd(m, n) = pm + qn$
    - By the Divisibility of Linear Combinations Theorem, since $d \mid a$ and $d \mid b$ (by assumption), we know that $d \mid pm + qn$
    - Therefore, $d \mid \gcd(m, n)$ QED

6.6 Proofs and Algorithms II: Computing the Greatest Common Divisor

Naively Searching for the GCD

- We had an implementation of the predicate as a function called divides

```
def divides(d: int, n: int) -> bool:
    """Return whether d divides n."""
    if d == 0:
        return n == 0
    else:
        return n % d == 0
```

- With the above function, we can implement a gcd function

```
def naive_gcd(m: int, n: int) -> int:
```

```
"""Return the gcd of m and n."""
if m == 0 and n == 0:
        return 0
else:
        possible_divisors = range(1, min(abs(m), abs(n) + 1))
        return max({d for d in possible_divisors if divides(d, m) and
        divides(d, n)
```

GCD and Remainders

- **Theorem.** (Quotient-Remainder Theorem) For all $n \in \mathbb{Z}$ and $d \in \mathbb{Z}$, if $d \neq 0$ then there exist $q \in \mathbb{Z}$ and $r \in \mathbb{N}$ such that $n = qd + r$ and $0 \leq r < |d|$. Moreover, these q and r are *unique* for a given n and d
- We say that q is the **quotient** when n is divided by d, and that r is the **remainder** when n is divided by d, and write $r = n \% d$
- **Theorem.** For all $a, b \in \mathbb{Z}$ where $b \neq 0$, $\gcd(a, b) = \gcd(b, a \% b)$
    - $Translation.$ $\forall a, b \in \mathbb{Z}, b \neq 0 \Longrightarrow \gcd(a, b) = \gcd(b, a \% b)$
    - $Discussion.$ We'll define the variable $d = \gcd(b, a \% b)$, and prove that $d = \gcd(a, b)$ as well. To do so, we'll need to prove that d divides both a and b, and that it is greater than every other common divisor of a and b
    - $Proof.$ Let $a, b \in \mathbb{Z}$ and assume $b \neq 0$. Let $r = a \% b$ (the remainder when a is divided by b). We need to prove that $\gcd(a, b) = \gcd(b, r)$
    - Let $d = \gcd(b, r)$. We'll prove that $d = \gcd(a, b)$ as well, by proving three things: that $d \mid a$, that $d \mid b$, and that every common divisor of a and b is $\leq d$
    - **Part 1:** proving that $d \mid a$
    - By our definition of r and the Quotient-Remainder Theorem, there exists $q \in \mathbb{Z}$ such that $a = qb + r$. Since $d = \gcd(b, r)$, d divides both b and r. By the Divisibility of Linear combinations Theorem, $d \mid qb + r$, and so $d \mid a$
    - **Part 2:** proving that $d \mid b$
    - By the definition of gcd, $d \mid b$
    - **Part 3:** proving that every common divisor of a and b is $\leq d$
    - Let $d_1 \in \mathbb{Z}$ and assume that $d_1 \mid a$ and $d_1 \mid b$. We will prove that $d_1 \leq d$
    - First, we'll prove that $d_1 \mid r$. By the Quotient-Remainder Theorem, the equation $a = qb + r$ can be rewritten as $r = a - qb$. Then using our assumption that d_1 is a common divisor of a and b, by the Divisibility of Linear Combinations Theorem, $d_1 \mid r$
    - So then $d_1 \mid b$ (by our assumption), and $d_1 \mid r$, and so it is a common divisor of b and r. Therefore by the definition of gcd, $d_1 \leq \gcd(b, r) = d$ QED

GCD, Remainders, and a New Algorithm

- The theorem above suggests a possible way of computing the gcd of 2 numbers in an iterative (repeated) fashion
    - i.e. 24 and 16
        - Since 24 % 16 = 8, gcd(24, 16) = gcd(16, 8)
        - Since 16 % 8 = 0, gcd(16, 8) = gcd(8, 0)
        - Since the gcd of any positive integer n and 0 is simply n itself, gcd(8, 0) = 8
        - gcd(24, 16) = 8
- The above algorithm of computing the gcd of 2 numbers is known as the *Euclidean algorithm*
- Euclidean Algorithm
    - *Given*: integers a and b. *Returns*: gcd(a, b)
    - 1. Initialize 2 variables x, y to the given numbers a and b
    - 2. Let r be the remainder when x is divided by y
    - 3. Reassign x and y to y and r, respectively
    - 4. Repeat steps 2 and 3 until y is 0
    - 5. At this point, x refers to the gcd of a and b
- Visualizing the changing values of x and y

Iteration	x	y
0	24	16
1	16	8
2	8	0

The While Loop

- A while loop looks similar to an if statement

```
while <condition>:
    <statement>
    …
```

    - Unlike an if statement, after executing its body the while loop will check the condition again. If the condition still evaluates to True, then the body is repeated
- Example

```
>>> numbers = []
>>> number = 1
>>> while number < 100:
. . .     numbers.append(number)
. . .     number = number * 2
. . .
```

```
>>> numbers
[1, 2, 4, 8, 16, 32, 64]
```

- number appear in both the while loop's body and its condition
- number increases at each iteration
- Eventually, number refers to the value 128 and the while loop is done because 128 < 100 evaluates to False
- The number of iterations is dependent on the initial value of number

Implementing the Euclidean Algorithm

```
def euclidean_gcd(a: int, b: int) -> int:
    """Return the gcd of a and b."""
    # Step 1: initialize x and y
    x = a
    y = b
    while y != 0:  # Step 4: repeat Steps 2 and 3 until y is 0
        # Step 2: calculate the remainder of x divided by y
        r = x % y

        # Step 3: reassign x and y
        x = y
        y = r

    # Step 5: x now refers to the gcd of a and b
    return x
```

- Step 1, initializing x and y, occurs in the code before the while loop begins
- Steps 2 and 3 are performed inside the loop body
- Step 4, the repetition, is achieved by the while loop. Rather than specifying a *stopping condition*, we must write a *continuing condition* (the negation of the stopping condition)
  - "until y = 0" becomes "while y != 0"
- Step 5, the return value, is exactly what is specified by the algorithm

Iteration	x	y
0	24	16
1	16	8
2	8	0

- We don't have a typical accumulator pattern
  - Both x and y are *loop variables* for the while loop

- In a for loop, the loop variable is initialized and reassigned automatically by the Python interpreter to each element of the collection being looped over
- In a while loop, the loop variable(s) must be initialized and reassigned explicitlyin code that we write
- Use for loops where possible (when we have an explicit collection to loop over) and reserve while loops for situations that can't be easily implemented with a for loop
- Parallel assignment
    - Suppose we had swapped the last two lines of the loop body

```
while y != 0:
    r = x % y
    y = r
    x = y
```

    x = y assigns x to the *new value* of y rather than its old one
    - *Parallel assignment* – a feature in which multiple variables can be assigned in the same statement

```
while y != 0:
    r = x % y
    x, y = y, r
```

    - The above assignment statement is evaluated as follows:
        - First, the right side y, r is evaluated, producing two objects
        - Then, each object is assigned to the corresponding variable on the left side
    - The right side is fully evaluated before any variable reassignment occurs
        - Order does not matter
    - Rewriting euclidean_gcd using parallel assignment:

```
def euclidean_gcd(a: int, b: int) -> int:
    """Return the gcd of a and b."""
    x, y = a, b

    while y != 0:
        r = x % y
        x, y = y, r

    return x
```

- Documenting loop properties: loop invariants
    - The Euclidean Algorithm relies on a key property – gcd(x, y) == gcd(y, x % y)
        - Even though x and y change, their gcd doesn't
    - gcd(x, y) == gcd(a, b)

        - This statement is called a *loop invariant*
        - *Loop invariant* – a property about loop variables that must be true at the start and end of each loop iteration
    - By convention, we document loop invariants at the top of a loop body using an assert statement

```
def euclidean_gcd(a: int, b: int) -> int:
    """Return the gcd of a and b."""
    x, y = a, b

    while y != 0:
        # assert naive_gcd(x, y) == naive_gcd(a, b)  # loop invariant
        r = x % y
        x, y = y, r

    return x
```

    - After the loop stops, the loop invariant should tell us that gcd(x, 0) == gcd(a, b), and so we know that x == gcd(a, b), which is why x is returned
    - To know for sure whether a loop invariant is correct, we need a proof

6.7 Modular Arithmetic

Intro

- $Definition.$ Let $a, b, n \in \mathbb{Z}$, and assume $n \neq 0$. We say that a **is equivalent to** b **modulo** n when $n \mid a - b$. In this case, we write $a \equiv b \pmod{n}$
    - a and b have the *same remainder* when divided by n
- **Theorem.** For all $a, b, c, d, n \in \mathbb{Z}$, if $n \neq 0$, if $a \equiv c \pmod{n}$ and $b \equiv d \pmod{n}$, then:
        - 1. $a + b \equiv c + d \pmod{n}$
        - 2. $a - b \equiv c - d \pmod{n}$
        - 3. $ab \equiv cd \pmod{n}$
    - $Translation\ 1.$

$$\forall a, b, c, d, n \in \mathbb{Z}, \big(n \neq 0 \wedge (n \mid a - c) \wedge (n \mid b - d)\big) \Longrightarrow n \mid (a + b) - (c + d)$$

    - $Proof\ 1.$ Let $a, b, c, d, n \in \mathbb{Z}$. Assume that $n \neq 0, n \mid a - c$, and $n \mid b - d$. This means we want to prove that $n \mid (a + b) - (c + d)$.
    - By the Divisibility of Linear Combinations Theorem, since $n \mid (a - c)$ and $n \mid (b - d)$, it divides their sum:

$$n \mid (a - c) + (b - d)$$
$$n \mid (a + b) - (c + d)$$

QED

- $Translation\ 2.$

$$\forall a, b, c, d, n \in \mathbb{Z}, \big(n \neq 0 \wedge (n \mid a - c) \wedge (n \mid b - d)\big) \Longrightarrow$$
$$n \mid (a - b) - (c - d)$$

- $Proof\ 2.$ Let $a, b, c, d, n \in \mathbb{Z}$. Assume that $n \neq 0, n \mid a - c$, and $n \mid b - d$. This means we want to prove that $n \mid (a - b) - (c - d)$.
- By the Divisibility of Linear Combinations Theorem, since $n \mid (a - c)$ and $n \mid (b - d)$, it divides their difference:

$$n \mid (a - c) - (b - d)$$
$$n \mid (a - b) - (c - d)$$

QED

- $Translation\ 3.$

$$\forall a, b, c, d, n \in \mathbb{Z}, \big(n \neq 0 \wedge (n \mid a - c) \wedge (n \mid b - d)\big) \Longrightarrow$$
$$n \mid ab - cd$$

- $Proof\ 3.$ Let $a, b, c, d, n \in \mathbb{Z}$. Assume that $n \neq 0, n \mid a - c$, and $n \mid b - d$. This means we want to prove that $n \mid ab - cd$.
- Expanding the definition of division, we want to show that

$$\exists k \text{ such that } ab - cd = kn$$

- Take $k = n(q_a q_b - q_c q_d) + r_1(q_b - q_d) + r_2(q_a - q_c)$
- By the Quotient-Remainder Theorem,

$$a = nq_a + r_1$$
$$c = nq_c + r_1$$
$$b = nq_b + r_2$$
$$d = nq_d + r_2$$
$$ab = n^2 q_a q_b + nr_1 q_b + nr_2 q_a + r_1 r_2$$
$$cd = n^2 q_c q_d + nr_1 q_d + nr_2 q_c + r_1 r_2$$
$$ab - cd = n^2(q_a q_b - q_c q_d) + nr_1(q_b - q_d) + nr_2(q_a - q_c) = kn$$

QED

<u>Modular Division</u>

- Division does *not* preserve modular equivalence
- **Theorem.** (*Modular inverse*) Let $n \in \mathbb{Z}^+$ and $a \in \mathbb{Z}$. If $\gcd(a, n) = 1$, then there exists $p \in \mathbb{Z}$ such that $ap \equiv 1 \pmod{n}$.
    - We call this p a **modular inverse of** a **modulo** n
    - $Translation.\ \forall n \in \mathbb{Z}^+, \forall a \in \mathbb{Z}, \gcd(a, n) = 1 \Longrightarrow (\exists p \in \mathbb{Z}, ap \equiv 1 \pmod{n})$
    - $Proof.$ Let $n \in \mathbb{Z}^+$ and $a \in \mathbb{Z}$. Assume $\gcd(a, n) = 1$
    - Since $\gcd(a, n) = 1$, by the GCD Characterization Theorem we know that there exist integers p and q such that $pa + qn = \gcd(a, n) = 1$

- Rearranging the equation, we get that $pa - 1 = -qn$, and so (by the definition of divisibility, taking $k = -q$), $n \mid pa - 1$
- Then by the definition of modular equivalence, $pa \equiv 1 \pmod{n}$ QED

- **Ex.** Let $a \in \mathbb{Z}$ and $n \in \mathbb{Z}^+$. If $\gcd(a, n) = 1$, then for all $b \in \mathbb{Z}$, there exists $k \in \mathbb{Z}$ such that $ak \equiv b \pmod{n}$
  - $Translation.$ $\forall a, n \in \mathbb{Z}, \gcd(a, n) = 1 \Longrightarrow (\forall b \in \mathbb{Z}, \exists k \in \mathbb{Z}, ak \equiv b \pmod{n})$
  - $Discussion.$ This is saying that under the given assumptions, b is "divisible" by a modulo p.
  - Since it is assumed that $\gcd(a, n) = 1$, we can use the modular inverses theorem, which gives us a $p \in \mathbb{Z}$ such that $ap \equiv 1 \pmod{n}$
    - Looks like we can multiply both sides by b
  - $Proof.$ Let $a \in \mathbb{Z}$ and $n \in \mathbb{Z}^+$. Assume $\gcd(a, n) = 1$, and let $b \in \mathbb{Z}$. We want to prove that there exists $k \in \mathbb{Z}$ such that $ak \equiv b \pmod{n}$
  - First, using the Modular Inverses theorem, since we assumed $\gcd(a, n) = 1$, there exists $p \in \mathbb{Z}$ such that $ap \equiv 1 \pmod{n}$
  - Second, since modular equivalence preserves multiplication, $apb \equiv b \pmod{n}$
  - Let $k = pb$, we have that $ak \equiv b \pmod{n}$ QED

<u>Exponentiation and Order</u>

- Powers of positive integers increase without bound
- Because there are only a finite number of remainders for any given $n \in \mathbb{Z}^+$, for any $a \in \mathbb{Z}$ the infinite sequence of *remainders* of $a^0, a^1, a^2, a^3, \ldots$ must repeat at some point
  - For example, let's see what happens for each of the possible bases modulo 7:
    - $0: 0^1 \equiv 0 \pmod{7}, 0^2 \equiv 0 \pmod{7}$
    - $1: 1^1 \equiv 1 \pmod{7}, 1^2 \equiv 1 \pmod{7}$
    - $2: 2^1 \equiv 2 \pmod{7}, 2^2 \equiv 4 \pmod{7}, 2^3 \equiv 1 \pmod{7}, 2^4 \equiv 2 \pmod{7}$
    - $3: 3^1 \equiv 3 \pmod{7}, 3^2 \equiv 2 \pmod{7}, 3^3 \equiv 6 \pmod{7},$ $3^4 \equiv 4 \pmod{7}, 3^5 \equiv 5 \pmod{7}, 3^6 \equiv 1 \pmod{7}, 3^7 \equiv 3 \pmod{7}$
    - $4: 4^1 \equiv 4 \pmod{7}, 4^2 \equiv 2 \pmod{7}, 4^3 \equiv 1 \pmod{7}, 4^4 \equiv 4 \pmod{7}$
  - No matter which number we start with, we enter a cycle
    - i.e. the cycle starting with 2 is [2, 4, 1, 2, …]
      - This cycle has length 3, since it takes 3 elements in the sequence for the 2 to repeat
  - Cycle lengths for each possible $a \in \{0, 1, \ldots, 6\}$:

a	Cycle length
0	1
1	1
2	3

3	6
4	3
5	6
6	2

  - For each base other than 0, the cycle length for base a is the smallest positive integer k such that $a^k \equiv 1 \pmod 7$
    - i.e. $2^3 \equiv 1 \pmod 7$, and the cycle repeats at $2^4 \equiv 2^3 \cdot 2 \equiv 2 \pmod 7$
  - Cycle length for the remainders modulo 7 always divide 6
- $Definition.$ Let $a \in \mathbb{Z}$ and $n \in \mathbb{Z}^+$. We define the **order of a modulo n** to be the smallest positive integer k such that $a^k \equiv 1 \pmod n$, when such a number exists
  - We denote the order of a modulo n as $\mathrm{ord}_n(a)$
- The table for modulo 17

a	Cycle length
0	1
1	1
2	8
3	16
4	4
5	16
6	16
7	16
8	8
9	8
10	16
11	16
12	16
13	4
14	16
15	8
16	2

  - The cycle length for these bases always divides 16
  - Again, for each base a other than 1, the cycle length corresponding to a is the least positive integer k such that $a^k \equiv 1 \pmod{17}$
- For 17, *every* base a other than 0 satisfies $a^{16} \pmod{17}$
  - This generalizes to every prime number
- **Theorem.** (*Fermat's Little Theorem*) Let $p, a \in \mathbb{Z}$ and assume p is prime and that $p \nmid a$. Then $a^{p-1} \equiv 1 \pmod p$

Euler's Theorem

- $Definition.$ We define the function $\varphi: \mathbb{Z}^+ \to \mathbb{N}$, called the **Euler totient function** (or **Euler phi function**), as follows:

$$\varphi(n) = |\{\, a \mid a \in \{1, \ldots, n-1\}, \text{and} \gcd(a, n) = 1 \,\}|$$

  - Examples of the Euler totient function:
    - $\varphi(5) = 4$, since {1, 2, 3, 4} are all coprime to 5
    - $\varphi(6) = 2$, since only {1, 5} are coprime to 6
    - In general, for any prime number $p, \varphi(p) = p - 1$, since all the numbers {1, 2, …, $p - 1$} are coprime to p
    - $\varphi(15) = 10$, since the numbers {1, 2, 4, 7, 8, 11, 13, 14} are all coprime to 15.
      - Note that "removed" numbers are all multiples of 3 or 5, the prime factors of 15
    - In general, for any two distinct primes p and q, $\varphi(pq) = (p-1)(q-1)$, although this is not obvious, and requires a proof
- With the Euler totient function in hand, we can now state the generalization of Fermat's Little Theorem
- **Theorem.** (*Euler's Theorem*). For all $a \in \mathbb{Z}$ and $n \in \mathbb{Z}^+$, if $\gcd(a, n) = 1$, then $a^{\varphi(n)} \equiv 1 \pmod{n}$

7.1 An Introduction to Cryptography

Cryptography

- *Cryptography* – study of theoretical and practical techniques for keeping data secure
- Encryption involves turning coherent messages into seemingly-random nonsensical strings, and then back again

Setting the Stage: Alice and Bob

- *Two-party confidential communication* – simplest setup in cryptography
  - We have two people, Alice and Bob, who wish to send messages to each other that only they can read
  - We also have Eve, who has access to all of the communications between Alice and Bob, and wants to discover what they're saying
- Through some shared piece of information called a secret key, Alice and Bob need to encrypt their messages in such a way that they will each be able to decrypt each other's messages
  - Eve won't be able to decrypt the messages without knowing their secret key

- We define a *secure symmetric-key cryptosystem* as a system with the following parts:
    - A set P of possible original messages, called the *plaintext* messages
        - E.g. a set of strings
    - A set C of possible encrypted messages, called the *ciphertext*, messages
        - E.g. another set of strings
    - A set K of possible *shared secret keys*
        - Known by both Alice and Bob, but no one else
    - Two functions $Encrypt : K \times P \to C$ and $Decrypt : K \times P \to C$ that satisfies the following two properties:
        - *Correctness* – For all $k \in K$ and $m \in P$, $Decrypt\big(k, Encrypt(k, m)\big) = m$
            - i.e. if we encrypt and encrypt the same message with the same key, we get back the original message
        - *Security* – For all $k \in K$ and $m \in P$, if an eavesdropper only knows the value of $c = Encrypt(k, m)$ but does not know k, it is computationally infeasible to find m

<u>Example: Caesar's Substitution Cipher</u>

- The plaintext and ciphertext sets are strings, and the secret key is some positive integer k
- Consider messages that consist of uppercase letters and spaces, and associate each letter with a number as follows:

Character	Value	Character	Value	Character	Value	Character	Value
'A'	0	'H'	7	'O'	14	'V'	21
'B'	1	'I'	8	'P'	15	'W'	22
'C'	2	'J'	9	'Q'	16	'X'	23
'D'	3	'K'	10	'R'	17	'Y'	24
'E'	4	'L'	11	'S'	18	'Z'	25
'F'	5	'M'	12	'T'	19	' '	26
'G'	6	'N'	13	'U'	20		

    - In Python, we can implement this conversion as follows:

```
LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ '

def letter_to_num(c: str) -> int:
    """Return the number that corresponds to the given letter.

    Preconditions:
        - len(c) == 1 and c in LETTERS
    """
```

```
        return str.index(LETTERS, c)


    def num_to_letter(n: int) -> str:
        """Return the letter that corresponds to the given number.

        Preconditions:
            - 0 <= n < len(LETTERS)
        """
        return LETTERS[n]
```

- In the Caesar cipher, the secret key k is an integer from the set {1, 2, …, 26}
- Before sending any messages, Alice and Bob meet and decide on a secret key from the set
- When Alice wants to send a string message m to Bob, *encrypts* her message as follows:
    - For each letter of m, Alice shifts it by adding the secret key k to its corresponding numbers, taking remainders modulo 27, the length of LETTERS
    - i.e. if $k = 3$, and the plaintext message is 'HAPPY', encryption happens as follows:

Plaintext Character	Corresponding Integer	Shifted Integer	Ciphertext Character
'H'	7	10	'K'
'A'	0	3	'D'
'P'	15	18	'S'
'P'	15	18	'S'
'Y'	24	0	'A'

    - When Bob receives the ciphertext 'KDSSA', he decrypts the ciphertext by applying the corresponding shift in reverse
        - i.e. subtracting the secret key k instread of adding it
    - We can implement the above example in Python

```
def encrypt_caesar(k: int, plaintext: str) -> str:
    """Return the encrypted message using the Caesar cipher with key k.

    Preconditions:
        - all({x in LETTERS for x in plaintext})
        - 1 <= k <= 26
    """
    l = len(LETTERS)
    ciphertext = ''
```

```
        for letter in plaintext:
            ciphertext += num_to_letter((letter_to_num(letter) + k) % l))

        return ciphertext


    def decrypt_caesar(k: int, ciphertext: str) -> str:
        """Return the decrypted message using the Caesar cipher with key k.

        Preconditions:
            - all({x in LETTERS for x in plaintext})
            - 1 <= k <= 26
        """
        l = len(LETTERS)
        plaintext = ''

        for letter in ciphertext:
            plaintext += num_to_letter((letter_to_num(letter) – k) % l))

        return plaintext
```

Expanding the Set of Letters

- The key mathematical idea of Caesar generalizes to larger sets of letters
- Recall the following two built-in Python functions

```
>>> ord('A')  # Convert a character into an integer
65
>>> chr(33)  # Convert an integer into a character
'!'
```

- Using the above two functions, we can modify our encrypt and decrypt functions to operate on arbitrary Python strings
    - Our secret key now takes on values from the set {1, 2, …, 127}

```
def encrypt_ascii(k: int, plaintext: str) -> str:
    """Return the encrypted message using the Caesar cipher with key k.

    Preconditions:
        - all({ord(c) < 128 for c in plaintext})
        - 1 <= k <= 127
```

```
        """
        ciphertext = ''

        for letter in plaintext:
            ciphertext += chr((ord(letter) + k) % 128)

        return ciphertext


    def decrypt_caesar(k: int, ciphertext: str) -> str:
        """Return the decrypted message using the Caesar cipher with key k.

        Preconditions:
            - all({ord(c) < 128 for c in plaintext})
            - 1 <= k <= 127
        """
        plaintext = ''

        for letter in ciphertext:
            plaintext += chr((ord(letter) - k) % 128)

        return plaintext
```

- The Caesar cipher is not secure
    - An eavesdropper can try all possible secret keys to decrypt a ciphertext

7.2 The One-Time Pad and Perfect Secrecy

<u>Intro</u>

- The Caesar cipher should never be used in practice
    - Consider the ciphertext '0LaT0+T^+NZZW'
        - The 1st and the 5th letters in the plaintext must be the same
        - The 1st and 10th characters of the plaintext must be consecutive ASCII characters
    - Vulnerable to a *brute-force exhaustive key search attack*
        - Given a ciphertext, it is possible to try out every secret key and see which key yields a meaningful plaintext message

<u>The One-Time Pad</u>

- Works by shifting each character in the plaintext message, but the shift is *not* the same for each character
- Uses a string of length greater than or equal to the length of the plaintext message we wish to encrypt
- To *encrypt* a plaintext ASCII message m with a secret key k, for each index i between 0 and $|m| - 1$, we compute:
    - $(m[i] + k[i])\ \%\ 128$, where $m[i]$ and $k[i]$ are converted to their numeric representations to do the arithmetic
- Ex. encrypting the plaintext 'HELLO' with the secret key 'david'

Plaintext			Key			Ciphertext	
H	72		d	100		,	44
E	69	+	a	97	→	&	38
L	76		v	118		B	66
L	76		i	105		5	53
O	79		d	100		3	51

- For decryption, we take the ciphertext c and recover the plaintext by subtracting each letter of the secret key:
    - $(c[i] - k[i])\ \%\ 128$

<u>Perfect Secrecy and Its Costs</u>

- The one-time pad cryptosystem has the property of *perfect secrecy*
    - *Perfect secrecy* – a ciphertext reveals no information about its corresponding plaintext other than its length
    - i.e. *any* five-letter plaintext message can be encrypted to obtain ',&B53'
- Because of perfect secrecy, an eavesdropper cannot gain any information about the original plaintext message
- Cost: the secret key must have at least the same length as the message being sent, and cannot be reused from one message to another

<u>Stream Ciphers</u>

- *Stream cipher* – a type of symmetric-key cryptosystem that emulate a one-time pad but share a much smaller secret key
- The shared secret key is small, and both parties use an algorithm to generate an arbitrary number of new random characters, based on both the secret key and any previously-generated characters
- Do not have perfect secrecy, since the characters used in encryption aren't truly random, though it can appear "random" if the algorithm is good enough

7.3. Computing Shared Secret Keys

Alice and Bob are Mixing Paint

- Suppose that Alice and Bob would like to establish a secret *paint colour* that only the two of them know, they would use the following procedure
- Step 1. They both agree on a random, not-secret colour of paint to start with
    - i.e. yellow
    - Eavesdroppers also know this colour
- Step 2. They each choose their own secret colour, which they will never share with each other or anyone else
    - i.e. Alice chooses red and Bob chooses teal
- Step 3. They each mix their secret colours with their shared colour yellow
    - i.e. Alice gets light orange, Bob gets blue
    - This is done in secret
- Step 4. They exchange these colours with each other
    - This is done publicly
    - At this point, there are 3 not-secret colours: yellow and the two mixtures
    - There are 2 secret colours: Alice's red and Bob's teal
- Step 5. Alice mixes Bob's blue colour with her original secret red to produce a brown. Bob mixes Alice's light orange with his original secret teal to produce the same brown.
    - They are the same brown because they both consist of the same mixture of three colours:
        - Yellow (shared)
        - Red (Alice's secret)
        - Teal (Bob's secret)
- Any eavesdropper has access to 3 colours: the original shared yellow, and the two mixtures orange and blue
    - Assume colour mixtures are not easily separated, the eavesdropper cannot determine what Alice and Bob's secret colours are, and therefore can't mix them together with the yellow to produce the right shade of brown

The Diffie-Hellman Key Exchange

- Setting: two parties, Alice and Bob
- Result: Alice and Bob share a secret key k
- Step 1. Alice chooses a prime number p greater than 2 and an integer g which satisfies $2 \leq g \leq p-1$, and sends both to Bob
- Step 2. Alice chooses a secret number $a \in \{1, 2, .., p-1\}$ and sends Bob $A = g^a \ \% \ p$
- Step 3. Bob chooses a secret number $b \in \{1, 2, \dots, p-1\}$ and sends Alice $B = g^b \ \% \ p$

- Step 4. Alice computes $k_A = B^a \% p$. Bob computes $k_B = A^b \% p$
    - It turns out that $k_A = k_B$, and so this value is chosen as the secret key k that Alice and Bob share
- Example
    - 1. Alice starts by choosing $p = 23$ and $g = 2$. She sends both p and g to Bob
    - 2. Alice chooses a secret number $a = 5$. She sends $A = g^a \% p = 2^5 \% 23 = 9$ to Bob
    - 3. Bob chooses a secret number $b = 14$. He sends $B = g^b \% p = 2^{14} \% 23 = 8$ to Alice
    - 4. Alice computes $k_A = B^a \% p = 8^5 \% 23 = 16$. Bob computes $k_B = A^b \% p = 9^{14} \% 23 = 16$. $k_A = k_B$, and so these form the secret key k

Correctness: Are k_A and k_B Always Equal?

- **Theorem.** (Correctness of Diffie-Hellman key exchange)

$$\forall p, g, a, b \in \mathbb{Z}^+, (g^b \% p)^a \% p = (g^a \% p)^b \% p$$

- $Discussion$. We can analyze the numbers using modular arithmetic modulo p. In this case the calculation involves just switching around exponents in g^{ab}
- $Proof$. Let $p, g, a, b \in \mathbb{Z}^+$. Let $A = g^a \% p$ and $B = g^b \% p$. We will prove that $B^a \% p = A^b \% p$
- First, we have that $A \equiv g^a \pmod p$ and $B \equiv g^b \pmod p$. So then $A^b \equiv (g^a)^b \equiv g^{ab} \pmod p$, and $B^a \equiv (g^b)^a \equiv g^{ab} \pmod p$. Since $g^{ab} = g^{ba}$, we can conclude that $A^b \equiv B^a \pmod p$
- So then A^b and B^a must have the same remainder when divided by p, and so $B^a \% p = A^b \% p$

Security: How Secret is the Key?

- Over the course of the algorithm, the eavesdropper has access to $p, g, g^a \% p, g^b \% p$
    - Question: from the information, can the eavesdropper determine the secret key k?
- The eavesdropper can try to compute a and b directly. This is an instance of the *discrete logarithm problem*
    - *Discrete logarithm problem* – given $p, g, y \in \mathbb{Z}^+$, find an $x \in \mathbb{Z}^+$ such that $g^x \equiv y \pmod p$
- While we could implement a *brute-force* algorithm for solving this problem that simply tries all possible exponents $x \in \{0, 1, \dots, p-1\}$, this is computationally inefficient in practice when p is chosen to be extremely large
- There is no known *efficient* algorithm for solving the discrete logarithm problem. Therefore, the Diffie-Hellman key exchange is *computationally secure*

- While there are known algorithms that eavesdroppers could use for determining the shared secret key, all known algorithms are computationally infeasible for standard primes chosen
- i.e. Diffie-Hellman key exchanges tend to use primes on the order of $2^{2048} \approx 10^{617}$

7.4 The RSA Cryptosystem

Intro

- Limitation of symmetric-key encryption: a secret key needs to be established for every pair of people who want to communicate
  - If there are n people who each want to communicate securely with each other, there are $\frac{n(n-1)}{2}$ keys needed
- *Public-key cryptosystem* – each person has two keys:
  - A private key known only to them
  - A public key known to everyone

Public-Key Cryptography

- Suppose Alice want to send Bob a message
  - Alice uses Bob's *public key* to encrypt the message
  - Bob uses his *private key* to decrypt the message
- A *secure public-key cryptosystem* has the following parts:
  - A set P of possible original messages, called *plaintext* messages (e.g. a set of strings)
  - A set C of possible encrypted messages, called *ciphertext* messages (e.g. another set of strings)
  - A set K_1 of possible public keys and a set K_2 of possible private keys
  - A subset $K \subseteq K_1 \times K_2$ of possible *public-private key pairs*
    - Not every public key can be paired with every private key
  - Two functions $Encrypt : K_1 \times P \to C$ and $Decrypt : K_2 \times C \to P$ that satisfy the following two properties:
    - *Correctness* – For all $(k_1, k_2) \in K$ and $m \in P$, $Decrypt\big(k_2, Encrypt(k_1, m)\big) = m$
      - i.e. if we encrypt and then decrypt the same message with a public-private key pair, we get back the original message
    - *Security* – For all $(k_1, k_2) \in K$ and $m \in P$, if an eavesdropper only knows the values of the public key k_1 and the ciphertext $c = Encrypt(k_1, m)$

but does not know k_2, it is computationally infeasible to find the plaintext message m

The RSA Cryptosystem

- The Diffie-Hellman key exchange algorithm relies on the hardness of the *discrete logarithm problem*
- The *Rivest-Shamir-Adleman (RSA) cryptosystem* relies on the hardness of factoring large integers (i.e. with hundreds of digits)
- Phase 1: Key Generation
    - Step 1. Alice picks two distinct prime numbers p and q
    - Step 2. Alice computes the product $n = pq$
    - Step 3. Alice chooses an integer $e \in \{2, 3, \dots, \varphi(n) - 1\}$ such that $\gcd\big(e, \varphi(n)\big) = 1$
    - Step 4. Alice chooses an integer $d \in \{2, 3, \dots, \varphi(n) - 1\}$ that is the modular inverse of e modulo $\varphi(n)$
        - i.e. $de \equiv 1 \pmod{\varphi(n)}$
    - Alice's *private key* is the tuple (p, q, d), and her public key is tuple (n, e)
- Phase 2: Message Encryption
    - Bob wants to send Alice a plaintext message m
    - Bob computes the ciphertext $c = m^e \,\%\, n$ and sends it to Alice
- Phase 3: Message Decryption
    - Alice computes $m' = c^d \,\%\, n$
- Example
    - 1. Alice chooses the primes numbers $p = 23$ and $q = 31$
    - 2. The product is $n = p \cdot q = 23 \cdot 31 = 713$
    - 3. Alice chooses an e where $\gcd\big(e, \varphi(n)\big) = 1$
        - Alice calculates that $\varphi(713) = 660$, and chooses $e = 547$ to satisfy the constraints on e
    - 4. Alice calculates the modular inverse to find out the last part of the private key
        - $d \cdot 547 = 1 \pmod{660}$, so $d = 403$
    - Private key: $(p = 23, q = 31, d = 403)$, public key: $(n = 713, e = 547)$
    - Bob wants to send the number 42 to Alice.
        - He computes the encrypted number to be $c = 42^e \,\%\, n = 42^{547} \,\%\, 713 = 106$
    - Alice receives the number 106 from Bob. She computes the decrypted number to be $m = 106^d \,\%\, 713 = 106^{403} \,\%\, 713 = 42$

Proving the Correctness of RSA

- **Theorem.** Let $(p, q, d) \in \mathbb{Z}^+ \times \mathbb{Z}^+ \times \mathbb{Z}^+$ be a private key and $(n, e) \in \mathbb{Z}^+ \times \mathbb{Z}^+$ be its corresponding public key as generated by "RSA Phase 1". Let $m, c, m' \in \{1, \dots, n-1\}$ be the original plaintext message, ciphertext, and decrypted message, respectively.
- Then $m' = m$
- $Proof$. Let $p, q, n, d, e, m, c, m' \in \mathbb{N}$ be defined as in the above definition of the RSA algorithm. We need to prove that $m' = m$
- From the definition of m', we know $m' \equiv c^d \pmod{n}$. From the definition of c, we know $c \equiv m^e \pmod{n}$. Putting these together, we have:
$$m' \equiv (m^e)^d \equiv m^{ed} \pmod{n}$$
- We know that $de \equiv 1 \pmod{\varphi(n)}$, i.e. there exists a $k \in \mathbb{Z}$ such that $de = k\varphi(n) + 1$
- We also know that since $\gcd(m, n) = 1$, by Euler's Theorem $m^{\varphi(n)} \equiv 1 \pmod{n}$
- Putting the above all together, we have
$$\begin{aligned} m' &\equiv m^{ed} \pmod{n} \\ &\equiv m^{k\varphi(n)+1} \pmod{n} \\ &\equiv \left(m^{\varphi(n)}\right)^k \cdot m \pmod{n} \\ &\equiv 1^k \cdot m \pmod{n} \quad \text{(by Euler's Theorem)} \\ &\equiv m \pmod{n} \end{aligned}$$
- So $m' \equiv m \pmod{n}$. Since we also know m and m' are between 1 and $n-1$, we can conclude that $m' = m$ QED

<u>The Security of RSA</u>

- We know from the RSA encryption phase that $c \equiv m^e \pmod{n}$, so if we know all 3 of c, e, n, can we determine the value of m?
  - *No!* We don't have an efficient way of computing "e-th roots" in modular arithmetic
- We can attempt to discover Alice's private key. Since $de \equiv 1 \pmod{\varphi(n)}$, d is the inverse of e modulo $\varphi(n)$
  - We can compute the modular inverse of e modulo $\varphi(n)$ when we know both e and $\varphi(n)$, but right now we only know n, not $\varphi(n)$
  - If $n = p \cdot q$ where p and q are distinct primes, then $\varphi(n) = (p-1)(q-1)$, however, it is *not computationally feasible* to factor n when it is extremely large
    - This is known as the *Integer Factorization Problem*
    - There is no known efficient general algorithm for factoring integers

7.5 Implementing RSA in Python

<u>Key Generation</u>

- Assume that prime numbers p and q are given

```
def rsa_generate_key(p: int, q: int) -> Tuple[Tuple[int, int, int], Tuple[int, int]]:
    """Return an RSA key pair generated using primes p and q.

    The return value is a tuple containing two tuples:
        1. The first tuple is the private key, containing (p, q, d).
        2. The second tuple is the public key, containing (n, e).

    Preconditions:
        - p and q are prime
        - p != q
    """
    # Compute the product of p and q
    n = p * q

    # Choose e such that gcd(e, phi_n) == 1
    phi_n = (p – 1) * (q – 1)

    # Since e is chosen randomly, we repeat the random choice
    # until e is coprime to phi_n.
    e = random.randint(2, phi_n – 1)
    while math.gcd(e, phi_n – 1)
        e = random.randint(2, phi_n – 1)

    # Choose d such that e * d % phi_n = 1
    # Notice that we're using our modular_inverse from our work in the last chapter
    d = modular_inverse(e, phi_n)

    return ((p, q, d), (n, e))
```

Encrypting and Decrypting a Number

- The plaintext is a number m between 1 and $n - 1$
- The ciphertext is another number $c = m^e \% n$

```
def rsa_encrypt(public_key: Tuple[int, int], plaintext: int) -> int:
    """Encrypt the given plaintext using the recipient's public key.

    Preconditions:
        - public_key is a valid RSA public key (n, e)
        - 0 < plaintext < public_key[0]
```

```
    """
    n, e = public_key

    encrypted = (plaintext ** e) % n

    return encrypted


def rsa_decrypt(private_key: Tuple[int, int, int], ciphertext: int) -> int:
    """Decrypt the given ciphertext using the recipient's private key.

    Preconditions:
        - private_key is a valid RSA private key (p, q, d)
        - 0 < ciphertext < private_key[0] * private_key[1]
    """
    p, q, d = private_key
    n = p * q

    decrypted = (ciphertext ** d) % n

    return decrypted
```

Encrypting and Decrypting Text Using RSA

- The above implementation of RSA is unsatisfying because it encrypts numbers instead of strings
- One strategy is to take a string and break it up into individual characters and encrypt each character
- We can use ord/chr to convert between characters and numbers
- We can use a string accumulator to keep track of the encrypted/decrypted results

```
def rsa_encrypt_text(public_key: Tuple[int, int], plaintext: str) -> str:
    """Encrypt the given plaintext using the recipient's public key.

    Preconditions:
        - public_key is a valid RSA public key (n, e)
        - all({0 < ord(c) < public_key[0] for c in plaintext})
    """
    n, e = public_key
```

```
        encrypted = ''
        for letter in plaintext:
            # Note: we could have also used our rsa_encrypt function here instead
            encrypted = encrypted + chr((ord(letter) ** e) % n)

        return encrypted


    def rsa_decrypt_text(private_key: Tuple[int, int, int], ciphertext: int) -> int:
        """Decrypt the given ciphertext using the recipient's private key.

        Preconditions:
            - private_key is a valid RSA private key (p, q, d)
            - all({0 < ord(c) < private_key[0] * private_key[1] for c in ciphertext})
        """
        p, q, d = private_key
        n = p * q

        decrypted = ''
        for letter in ciphertext:
            # Note: we could have also used our rsa_decrypt function here instead
            decrypt = decrypted + chr((ord(letter) ** d) % n)

        return decrypted
```

7.6 Application: Securing Online Communications

HTTPS and the Transport Layer Security Protocol

- The *HTTPS* protocol consists of two parts:
  - *HTTP (Hypertext Transfer Protocol)* – governs the format of data being exchanged between our computer and the server
  - *TLS (Transport Layer Security)* – governs how the data formatted by HTTP is encrypted during the communication process
- HTTP allows our computer to communicate with servers around the world
- When combined with TLS, those communications are secure and cannot be "snooped" by an eavesdropper

TLS: An Overview (Simplified)

- *Client* – refers to our computer
- *Server* – refers to the website we are communicating with
- When the client initiates a request to the server (e.g. when we type in a URL into our web browser and press "Enter")
    - Step 1. When the client initiates the request, the server sends a "proof of identity" that the client has connected with the intended server, which the client verifies
        - Not encrypted
    - Step 2. Then, the client and server perform the Diffie-Hellman key exchange algorithm to establish a shared secret key
        - Not encrypted
    - Step 3. All remaining communication (e.g. the actual website data) is encrypted using an agreed-upon symmetric-key cryptosystem, like a stream cipher
- Two key questions
    - 1. Why is symmetric-key encryption (rather than public-key encryption) used to encrypt the communication in step 3?
    - 2. Given that the first two steps of TLS are unencrypted, how can the client be sure it is actually communicating with the intended server the whole time?

<u>Why Symmetric-Key Encryption?</u>

- Public-key cryptosystems, like RSA, are significantly *slower* than their symmetric-key counterparts
    - RSA relies on modular exponentiation as the key encryption

<u>Who Am I Connected To?</u>

- Question: how do we know we are communicating with the right server?
    - i.e. are we connecting to the real Google server, not some fake server that's simply pretending to be Google?
- We need some way to verify that the server (e.g. Google) we intend to speak with is actually who they say they are
- Every public-key cryptosystem (including RSA) can implement two additional algorithms to:
    - Sign message using the private key
    - Verify a signature using the public key
- These algorithms allow a server to *sign every message it sends* with its private key, and then have the client *verify* each message signature using the server's public key
    - We call these *digital signatures*
    - They help us identify exactly who we are speaking with

- Alice can add her signature, which is a function of her private key, to a message. Bob can verify that Alice is the sender with Alice's public key
- Digital signatures are used in each of the first two steps in the TLS protocol
- Establishing identity: digital certificates
    - In the first step of TLS, Tthe "proof of identity" the server sends is called a *digital certificate*
        - *Digital certificate* – contains identifying information for the server, including:
            - Its domain (e.g. www.google.com)
            - Its organization name (e.g. "Google LLC")
            - Its *public key*
    - The certificate also includes the digital signature of a *certificate authority*
        - *Certificate authority* – an organization whose purpose is to issue digital certificates to website domains and verify the identities of the operators of each of those domains
    - When the client "verifies" the digital certificate provided by the server, the client is actually verifying the digital signature provided by the certificate authority, using the certificate authority's public key
- Maintaining identity during Diffie-Hellman
    - During the Diffie-Hellman algorithm, the server *signs all messages* it sends, so that at every step the client can verify that the message came from the intended server
        - This relies on the client knowing the server's public key, which it gets from the digital certificate in the previous step
    - It is this *digital signature* from the server that allows the client to consistently verify that it is communicating with the server, and that messages haven't been tampered with
    - At the end of Step 2, the client and server have a shared secret key, and can now communicate safely using symmetric-key encryption

<u>(In)effectiveness of Cryptography</u>

- Diffie-Hellman and RSA are secure because it is very difficult to extract the private part of the data from what is being public communicated
- Unfortunately, many servers use the same group of prime numbers
- Some steps of the Diffie-Hellman algorithm can be precomputed for a specific group of prime numbers
- 512-bit and 1024-bit keys are prone to the Logjam attack
    - 2048-bit keys are used to avoid it

8.1 An Introduction to Running Time

How Do We Measure Running Time?

- Consider the following function, which prints out the first n natural numbers:

```
def print_integers(n: int) -> None:
    for i in range(0, n):
        print(i)
```

- We expect the function to take longer as n gets larger
- Rather than use an empirical measurement of runtime, we use an abstract representation of runtime: the number of "basic operations" an algorithm executes
    - This means that we can analyze functions without needing a computer, and our analysis theoretically applies to any computer system
- "Basic operation" is a vague term:
    - What counts as a "basic operation"?
    - How do we tell which "basic operations" are used by an algorithm?
    - Do all "basic operations" take the same amount of time?

Linear Running Time

- For the function print_integers, we know that:
    - The for loop will call print once per iteration
    - The loop iterates n times (i.e. with i taking on the values 0, 1, 2, …, n – 1)
- For an input n, there are n calls to print
- We say that the running time of print_integers on input n is n basic operations
- If we plot n against this measure running time, we obtain a line
- We say that print_integers has a *linear* running time, as the number of basic operations is a linear function of the input n

Quadratic Running Time

- Consider a function that prints all combinations of pairs of integers:

```
def print_pairs(n: int) -> None:
    """Print all combinations of pairs of the first n natural numbers."""
    for i in range(0, n):
        for j in range(0, n):
            print(i, j)
```

- The outer loop repeats its body n times, and its body is another loop, which repeats *its* body n times
    - print is called n^2 times in total

- We say that print_pairs has a *quadratic* running time, as the number of basic operations is a quadratic function of the input n

Logarithmic Running Time

- Consider the following function, which prints out the powers of two that are less than a positive integer n

```
def print_powers_of_two(n: int) -> None:
    """Print the powers of two that are less than n.

    Preconditions:
        - n > 0
    """
    for i in range(0, math.ceil(math.log2(n))):
        print(2 ** i)
```

- The number of calls to print is $\log_2 n$
- The running time of print_powers_of_two is *approximately*, but not exactly $\log_2 n$
- We say that print_powers_of_two has a *logarithmic* running time

Constant Running Time

- Consider this function

```
def print_ten(n: int) -> None:
    """Print n ten times."""
    for i in range(0, 10):
        print(n)
```

- The loop iterates 10 times regardless of what n is
- We say that print_ten has a *constant* running time
    - The number of basic operations is independent to the input size

Basic Operations

- From fastest to slowest:
    - Constant running time → logarithmic running time → linear running time → quadratic running time
- There are different ways of interpreting "basic operations", i.e.
    - We assign a variable at every loop iteration
    - print calls take longer than variable assignment
    - Calling the function is also an operation
    - This can get extremely complicated

- No matter how we interpret "basic operations" we know for sure that linear is faster than quadratic

8.2 Comparing Asymptotic Function Growth with Big-O

Four Kinds of Dominance

- We will mainly be concerned about functions mapping the natural numbers to the nonnegative real numbers
  - i.e. functions $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$
- We will look at the property of the function that describes the long-term (i.e. *asymptotic*) growth of a function
- $Definition.$ Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g is **absolutely dominated by** f when for all $n \in \mathbb{N}, g(n) \leq f(n)$
  - **Ex.** Let $f(n) = n^2$ and $g(n) = n$. Prove that g is absolutely dominated by f.
  - $Translation.$ Unpacking the definition, $\forall n \in \mathbb{N}, g(n) \leq f(n)$
  - $Proof.$ Let $n \in \mathbb{N}$. We want to show that $n \leq n^2$.
  - **Case 1:** Assume $n = 0$. In this case, $n^2 = n = 0$, so the inequality holds.
  - **Case 2:** Assume $n \geq 1$. In this case, we take the inequality $n \geq 1$ and multiply both sides by n to get $n^2 \geq n$, or equivalently $n \leq n^2$ QED
  - Absolute dominance is too strict for our purposes
    - i.e. if $g(n) \leq f(n)$ for every natural number except 5, then we can't say that g is absolutely dominated by f
    - $g(n) = 2n$ is not absolutely dominated by $f(n) = n^2$
      - $g(n) \leq f(n)$ everywhere except $n - 1$
- $Definition.$ Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g **is dominated by** f **up to a constant factor** when there exists a positive real number c such that for all $n \in \mathbb{N}, g(n) \leq c \cdot f(n)$
  - **Ex.** Let $f(n) = n^2$ and $g(n) = 2n$. Prove that g is dominated by f up to a constant factor
  - $Translation.$ Unpacking the definition, $\exists c \in \mathbb{R}^+, \forall n \in \mathbb{N}, g(n) \leq c \cdot f(n)$
  - $Discussion.$ We already saw that n is absolutely dominated by n^2, so if the n is multiplied by 2, then we should be able to multiply n^2 by 2 as well to get the calculation to work out
  - $Proof.$ Let $c = 2$, and let $n \in \mathbb{N}$. We want to prove that $g(n) \leq c \cdot f(n)$, or in other words, $2n \leq 2n^2$
  - **Case 1:** Assume $n = 0$. In this case, $2n = 0$ and $2n^2 = 0$, so the inequality holds
  - **Case 2:** Assume $n \geq 1$. Taking the assumed inequality $n \geq 1$ and multiplying both sides by $2n$ yields $2n^2 \geq 2n$, or equivalently $2n \leq 2n^2$ QED

- "Dominated by up to a constant factor" allows us to ignore multiplicative constants in our functions
  - i.e. $n, 2n, 10000n$ are all *equivalent* in the sense that each one dominates the other two up to a constant factor
- Consider the functions $f(n) = n^2$ and $g(n) = n + 90$. No matter how much we scale up f by multiplying it by a constant, $f(0)$ will always be less than $g(0)$, so we cannot say that g is dominated by f up to a constant factor

- $Definition.$ Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g **is eventually dominated by** f when there exists $n_0 \in \mathbb{R}^+$ such that $\forall n \in \mathbb{N}$, if $n \geq n_0$, then $g(n) \leq f(n)$
  - **Ex.** Let $f(n) = n^2$ and $g(n) = n + 90$. Prove that g is eventually dominated by f
  - $Translation.$ $\exists n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \implies g(n) \leq f(n)$
  - $Discussion.$ We need to argue that for "large enough" values of n, $n + 90 \leq n^2$. We could solve it using factoring or the quadratic formula, but we can also choose a large enough value of n_0
  - $Proof.$ Let $n_0 = 90$, let $n \in \mathbb{N}$. We want to prove that $n + 90 \leq n^2$

$$\begin{aligned} n + 90 &\leq n + n \quad \text{since } n \geq 90 \\ &= 2n \\ &\leq n \cdot n \quad \text{since } n \geq 2 \\ &= n^2 \end{aligned}$$

    QED
  - This definition allows us to ignore "small" values of n
    - Important for ignoring the influence of slow-growing terms in a function, which may affect the function values for "small" n, but eventually are overshadowed by the faster-growing terms
      - i.e. it took a while for the faster growth rate of n^2 to "catch up" to $n + 90$

- $Definition.$ Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that g **is eventually dominated by** f **up to a constant factor** when there exist $c, n_0 \in \mathbb{R}^+$, such that for all $n \in \mathbb{N}$, if $n \geq n_0$, then $g(n) \leq c \cdot f(n)$
  - In this case, we also say that g **is Big-O of** f, and write $g \in \mathcal{O}(f)$
  - We formally define $\mathcal{O}(f)$ to be the *set* of functions that are eventually dominated by f up to a constant factor:

$$\mathcal{O}(f) = \{g \mid g : \mathbb{N} \to \mathbb{R}^{\geq 0}, \text{and } \exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \implies g(n) \leq c \cdot f(n)\}$$

  - **Ex.** Let $f(n) = n^3$ and $g(n) = n^3 + 100n + 5000$. Prove that $g \in \mathcal{O}(f)$.
  - $Translation.$ $\exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \implies n^3 + 100n + 5000 \leq cn^3$

- $Discussion.$ g is neither eventually dominated by f nor dominated by f up to a constant factor.
  - Splitting up the inequality $n^3 + 100n + 5000 \leq cn^3$,
    - $n^3 \leq c_1 n^3$
    - $100n \leq c_2 n^3$
    - $5000 \leq c_3 n^3$
  - If we can make these three inequalities true, adding them together will give us our desired result (setting $c = c_1 + c_2 + c_3$)
  - There are multiple approaches to satisfy these inequalities
  - **Approach 1:** focus on choosing n_0
    - We can satisfy the three inequalities even if $c_1 = c_2 = c_3 = 1$
    - $n^3 \leq n^3$ is always true (so for all $n \geq 0$)
    - $100n \leq n^3$ when $n \geq 10$
    - $5000 \leq n^3$ when $n \geq \sqrt[3]{5000} \approx 17.1$
    - We can pick n_0 to be the largest of the lower bounds on n, $\sqrt[3]{5000}$, and then these inequalities will be satisfied
  - **Approach 2:** Pick c_1, c_2, c_3 to make the right sides large enough to satisfy the inequalities
    - $n^3 \leq c_1 n^3$ when $c_1 = 1$
    - $100n \leq c_2 n^3$ when $c_2 = 100$
    - $5000 \leq c_3 n^3$ when $c_3 = 5000$, *as long as* $n > 1$
- $Proof.$ (using approach 1) Let $c = 3$ and $n_0 = \sqrt[3]{5000}$. Let $n \in \mathbb{N}$, and assume that $n \geq n_0$. We want to show that $n^3 + 100n + 5000 \leq cn^3$
- First, we prove the below inequalities:
  - $n^3 \leq n^3$ (since the two quantities are equal)
  - Since $n \geq n_0 \geq 10$, we know that $n^2 \geq 100$, and so $n^3 \geq 100n$
  - Since $n \geq n_0$, we know that $n^3 \geq n_0^3 = 5000$
- Adding the above inequalities,

$$n^3 + 100n + 5000 \leq n^3 + n^3 + n^3 = cn^3$$

QED

- $Proof.$ (using approach 2) Let $c = 5101$ and $n_0 = 1$. Let $n \in \mathbb{N}$, and assume that $n \geq n_0$. We want to show that $n^3 + 100n + 5000 \leq cn^3$
- First, we prove the below inequalities:
  - $n^3 \leq n^3$ (since the two quantities are equal)
  - Since $n \in \mathbb{N}$, we know that $n \leq n^3$, and so $100n \leq 100n^3$
  - Since $1 \leq n$, we know that $1 \leq n^3$, and then multiplying both sides by 5000 gives us $5000 \leq 5000n^3$

- Adding the above inequalities,
  $$n^3 + 100n + 5000 \leq n^3 + 100n^3 + 5000n^3 = 5101n^3 = cn^3$$
  QED

8.3 Big-O, Omega, and Theta

Intro

- Big-O is not necessarily an exact description of growth
  - i.e. $n + 10 \in \mathcal{O}(n^{100})$ is not necessarily informative
- Big-O allows us to express *upper bounds* on the growth of a function
  - It does not allow us to distinguish between an upper bound that is tight and one that vastly overestimates the rate of growth

Omega and Theta

- $Definition.$ Let $f, g : \mathbb{N} \to R^{\geq 0}$. We say that g **is Omega of** f when there exist constants $c, n_0 \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}$, if $n \geq n_0$, then $g(n) \geq c \cdot f(n)$. In this case, we can also write $g \in \Omega(f)$
  - When $g \in \Omega(f)$, then f is a *lower* bound on the growth rate of g
    - i.e. $n^2 - n \in \Omega(n)$
- $Definition.$ Let $f, g : \mathbb{N} \to R^{\geq 0}$. We say that g **is Theta of** f when g is both Big-O of f and Omega of f. In this case, we can write $g \in \Theta(f)$, and say that f is on g
  - Equivalently, g is Theta of f when there exist constants $c_1, c_2, n_0 \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}$, if $n \geq n_0$ then $c_1 f(n) \leq g(n) \leq c_2 f(n)$
  - The "Theta bound" means that the two functions have the same approximate rate of growth, not that one is larger than the other
    - i.e. $10n + 5 \in \Theta(n), 10n + 5 \notin \Theta(n^2)$

A Special Case: $\mathcal{O}(1), \Omega(1)$, and $\Theta(1)$

- Consider the constant function $f(n) = 1$
  - If a function g is Big-O of this f, i.e. $g \in \mathcal{O}(f)$,
    - $\exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \Longrightarrow g(n) \leq c \cdot f(n)$
    - $\exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \Longrightarrow g(n) \leq c$ (since $f(n) = 1$)
  - There exists a constant c such that $g(n)$ is eventually always less than or equal to c
  - We say that such functions g are *asymptotically bounded* with respect to their input, and write $g \in \mathcal{O}(1)$ to represent this
- Similarly, we use $g \in \Omega(1)$ to express that functions are greater than or equal to some constant c

- $g(n) = \frac{1}{n+1}$ is $\mathcal{O}(1)$, but not $\Omega(1)$
- $g(n) = n^2$ is $\Omega(1)$, but not $\mathcal{O}(1)$

- $\Theta(1)$ refers to functions that are both $\mathcal{O}(1)$ and $\Omega(1)$

Properties of Big-O, Omega, and Theta

- Elementary functions
  - **Theorem.** For all $a, b \in \mathbb{R}^+$, the following statements are true
    - If $a > 1$ and $b > 1$, then $\log_a n \in \Theta(\log_b n)$
    - If $a < b$, then $n^a \in \mathcal{O}(n^b)$ and $n^a \notin \Omega(n^b)$
    - If $a < b$, then $a^n \in \mathcal{O}(b^n)$ and $a^n \notin \Omega(b^n)$
    - If $a > 1$, then $1 \in \mathcal{O}(\log_a n)$ and $1 \notin \Omega(\log_a n)$
    - $\log_a n \in \mathcal{O}(n^b)$ and $\log_a n \notin \Omega(n^b)$
    - If $b > 1$, then $n^a \in \mathcal{O}(b^n)$ and $n^a \notin \Omega(b^n)$
  - Progression of functions toward longer running times

1	Constant
$\log_2 n, \log_3 n, \dots, \log_{100} n$	Logarithms
$n^{0.000000000000001}$	Powers of n
$n^{0.5}$	
n	
n^2	
$n^{10000000000000}$	
1.0000000001^n	Exponentials
2^n	
100^n	

- Basic properties
  - **Theorem.** For all $f : \mathbb{N} \to \mathbb{R}^{\geq 0}, f \in \Theta(f)$
  - **Theorem.** For all $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}, g \in \mathcal{O}(f)$ if and only if $f \in \Omega(g)$
  - **Theorem.** For all $f, g, h : \mathbb{N} \to \mathbb{R}^{\geq 0}$:
    - If $f \in \mathcal{O}(g)$ and $g \in \mathcal{O}(h)$, then $f \in \mathcal{O}(h)$
    - If $f \in \Omega(g)$ and $g \in \Omega(h)$, then $f \in \Omega(h)$
    - If $f \in \Theta(g)$ and $g \in \Theta(h)$, then $f \in \Theta(h)$
- Operations on functions
  - $Definition.$ Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We can define the **sum of f and g** as the function $f + g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ such that
    - $\forall n \in \mathbb{N}, (f + g)(n) = f(n) + g(n)$
  - **Theorem.** For all $f, g, h : \mathbb{N} \to \mathbb{R}^{\geq 0}$, the following hold
    - If $f \in \mathcal{O}(h)$ and $g \in \mathcal{O}(h)$, then $f + g \in \mathcal{O}(h)$
    - If $f \in \Omega(h)$, then $f + g \in \Omega(h)$

- If $f \in \Theta(h)$ and $g \in \mathcal{O}(h)$, then $f + g \in \Theta(h)$

- Proof for the first statement
  - $Translation.$
    $$\forall f, g, h : \mathbb{N} \to \mathbb{R}^{\geq 0}, \big(f \in \mathcal{O}(h) \wedge g \in \mathcal{O}(h)\big) \Longrightarrow f + g \in \mathcal{O}(h)$$
  - $Discussion.$ Assuming $f \in \mathcal{O}(h)$ tells us that there exist positive real numbers c_1 and n_1 such that for all $n \in \mathbb{N}$, if $n \geq n_1$ then $f(n) \leq c_1 \cdot h(n)$. There similarly exist c_2 and n_2 such that $g(n) \leq c_2 \cdot h(n)$ whenever $n \geq n_2$.
    - Warning: we can't assume that $c_1 = c_2$ or $n_1 = n_2$, or any other relationship between these two sets of variables
  - We want to prove that there exist $c, n_0 \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}$, if $n \geq n_0$ then $f(n) + g(n) \leq c \cdot h(n)$
  - We can assume the following inequalities:
    - $f(n) \leq c_1 h(n)$
    - $g(n) \leq c_2 h(n)$
    - Adding the two given inequalities gives the third
    - We need to make sure that both given inequalities hold by choosing n_0 to be large enough, and let c be large enough to take into account both c_1 and c_2
  - $Proof.$ Let $f, g, h : \mathbb{N} \to \mathbb{R}^{\geq 0}$, and assume $f \in \mathcal{O}(h)$ and $g \in \mathcal{O}(h)$. By these assumptions, there exist $c_1, c_2, n_1, n_2 \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}$,
    - if $n \geq n_1$, then $f(n) \leq c_1 \cdot h(n)$, and
    - if $n \geq n_2$, then $g(n) \leq c_2 \cdot h(n)$
  - We want to prove that $f + g \in \mathcal{O}(h)$, i.e., that there exist $c, n_0 \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}$, if $n \geq n_0$ then $f(n) + g(n) \leq c \cdot h(n)$
  - Let $n_0 = \max\{n_1, n_2\}$ and $c = c_1 + c_2$. Let $n \in \mathbb{N}$, and assume that $n \geq n_0$. We now want to prove that $f(n) + g(n) \leq c \cdot h(n)$
  - Since $n_0 \geq n_1$ and $n_0 \geq n_2$, we know that n is greater than or equal to n_1 and n_2 as well. Then using the Big-O assumptions,
    - $f(n) \leq c_1 \cdot h(n)$
    - $g(n) \leq c_2 \cdot h(n)$
  - Adding these two inequalities,
  - $f(n) + g(n) \leq c_1 h(n) + c_2 h(n) = (c_1 + c_2)h(n) = c \cdot h(n)$ QED

- **Theorem.** For all $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ and for all $a \in \mathbb{R}^+$, $a \cdot f \in \Theta(f)$
- **Theorem.** For all $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{R}^{\geq 0}$, if $g_1 \in \mathcal{O}(f_1)$ and $g_2 \in \mathcal{O}(f_2)$, then $g_1 \cdot g_2 \in \mathcal{O}(f_1 \cdot f_2)$
  - The statement is still true if we replace Big-O with Omega or Theta

- **Theorem.** For all $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, if $f(n)$ is eventually greater than or equal to 1, then $\lfloor f \rfloor \in \Theta(f)$ and $\lceil f \rceil \in \Theta(f)$

- Properties from calculus
  - Our asymptotic notation of $\mathcal{O}, \Omega, \text{and } \Theta$ are concerned with comparing the *long-term behaviour* of two functions
    - i.e. the *limit* of the function as its input approaches infinity
  - Let $f : \mathbb{N} \to \mathbb{R}$ and $L \in \mathbb{R}$. We have the following definitions:
    - $\lim_{n\to\infty} f(n) = L : \forall \epsilon \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0 \Longrightarrow |f(n) - L| < \epsilon$
    - $\lim_{n\to\infty} f(n) = \infty : \forall M \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0 \Longrightarrow f(n) > M$
  - **Theorem.** for all $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$, if $g(n) \neq 0$ for all $n \in \mathbb{N}$, then the following statement hold:
    - If there exists $L \in \mathbb{R}^+$ such that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L$, then $g \in \Omega(f)$ and $g \in \mathcal{O}(f)$.
      - In other words, $g \in \Theta(f)$
    - If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f \in \mathcal{O}(g)$ and $g \notin \mathcal{O}(f)$
    - If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then $g \in \mathcal{O}(f)$ and $f \notin \mathcal{O}(g)$
  - The converse of the above statements is not true

8.4 Analyzing Algorithm Running Time

Intro

- Consider the following function

```
def print_items(lst: list) -> None:
    for item in lst:
        print(item)
```

- We will concentrate on how the *size of the input* influences the running time of a program
- We measure running time using asymptotic notation, and not exact expressions
- *Basic operation* – any block of code whose running time does not depend on the size of the input
  - Including assignment statements, arithmetic calculations, list/string indexing

The Runtime Function

- We expect print_items to take the same amount time on every list of length 100
- $Definition.$ Let func be an algorithm. For every $n \in \mathbb{N}$, we define the set $I_{func,n}$ to be the set of allowed inputs to func of size n

    - i.e. $I_{print\_items,100}$ is the set of all lists of length 100, $I_{print\_items,0}$ is the set containing just one input: the empty set
- For all $n \in \mathbb{N}$, every element of $I_{print\_items,n}$ has the *same* runtime when passed to print_items
- $Definition.$ Let func be an algorithm whose runtime depends *only* on its input size. We define the *running time function of func* as $RT_{func} : \mathbb{N} \to \mathbb{R}^{\geq 0}$, where $RT_{func}(n)$ is equal to the running time of func when given an input of size n
- Goal of a *running time analysis* for func is to find a function f such that $RT_{func}\ in\ \Theta(f)$
- Runtime analysis
    - 1. Identify the blocks of code which can be counted as a single basic operation
        - They don't depend on the input size
    - 2. Identify any loops in the code, which cause basic operations to repeat
        - We need to figure out how many times those loops run, based on the size of the input
    - 3. Use our observations from the previous two steps to come up with an expression for the number of basic operations used in the algorithm
        - i.e. find an exact expression for $RT_{func}(n)$
    - 4. Use the properties of asymptotic notation to find an elementary function f such that $RT_{func} \in \Theta\big(f(n)\big)$
- **Ex.** Consider the function print_items. We define input size to be the *number of items of the input list*. Perform a runtime analysis of print_items
    - $Running\ time\ analysis$. Let n be the length of the input list lst
    - Each iteration of the loop can be counted as a single operation, because nothing in it depends on the size of the input list
    - The running time depends on the number of loop iterations
    - Since this is a for loop over the lst argument, thus the total number of basic operations performed is n, and so the running time is $RT_{print\_items}(n) = n$, which is $\Theta(n)$ QED
- **Ex.** Analyse the running time of the following function.

```
def my_sum(numbers: List[int]) -> int:
    sum_so_far = 0

    for number in numbers:
        sum_so_far += number

    return sum_so_far
```

    - $Running\ time\ analysis$. Let n be the length of the input list
    - This function body consists of three statements

        - The assignment statement counts as 1 step
        - The for loop takes n steps: it has n iterations, and each iteration takes 1 step
        - The return statement counts as 1 step
    - The total running time is the sum of these three parts: $1 + n + 1 = n + 2$, which is $\Theta(n)$ QED

Nested Loops

- Count the number of repeated basic operations in a loop starting with the *innermost* loop and working our way out
- **Ex.** Consider the following function:

```
def print_sums(lst: list) -> None:
    for item1 in lst:
        for item2 in lst:
            print(item1 + item2)
```

  Perform a runtime analysis of print_sums.
    - $Running\ time\ analysis$. Let n be the length of lst
    - The inner loop runs n times, and each iteration is just a single basic operation
    - The outer loop runs n times, and each of its iterations take n operations
    - The total number of basic operations is $RT_{print\_sums}(n) =$
      $$\text{steps for the inner loop} \times \text{number of times inner loop is repeated}$$
      $$= n \times n$$
      $$= n^2$$
    - So the running time of this algorithm is $\Theta(n^2)$ QED
- **Ex.** Consider the following function:

```
def f(lst: List[int]) -> None:
    for item in lst:
        for i in range(0, 10):
            print(item + i)
```

  Perform a runtime analysis of this function.
    - $Running\ time\ analysis$. Let n be the length of the input list lst
    - The inner loop repeats 10 times, and each iteration is a single basic operation, for a total of 10 basic operations
    - The outer loop repeats n times, and each iteration takes 10 steps, for a total of $10n$ steps.
    - The running time of this function is $\Theta(n)$
    - Alternatively, the inner loop's running time does not depend on the number of items in the input list, so we can count it as a single basic operation

- The outer loop runs n times, and each iteration takes 1 step, for a total of n steps, which is $\Theta(n)$ QED

- **Ex.** Analyze the running time of the following function.

```
def combined(lst: List[int]) -> None:
    # Loop 1
    for item in lst:
        for i in range(0, 10):
            print(item + i)

    # Loop 2
    for item1 in lst:
        for item2 in lst:
            print(item1 + item2)
```

  - $Running\ time\ analysis$. Let n be the length of lst. We have already seen that the first loop runs in time $\Theta(n)$, while the second loop runs in time $\Theta(n^2)$
  - By the Sum of functions theorem from the previous section, we can conclude that combined runs in time $\Theta(n^2)$. QED

Loop Iterations With Changing Costs

- **Ex.** Analyze the running time of the following function.

```
def all_pairs(lst: List[int]) -> None:
    for i in range(0, len(lst)):
        for j in range(0, i):
            print(lst[i] + lst[j])
```

  - $Discussion$. The inner loop's running time depends on the current value of i
  - We need to manually add up the cost of each iteration of the outer loop, which depends on the number of iterations of the inner loop
  - Since j goes from 0 to i – 1, the number of iterations of the inner loop is i, and each iteration of the inner loop counts as one basic operation
  - $Running\ time\ analysis$. Let n be the length of the input list.
  - We start by analysing the running time of the inner loop for a *fixed* iteration of the outer loop, and a fixed value of i
    - The inner loop iterates i times and each iteration takes one step.
    - Therefore the cost of the inner loop is i steps, for one iteration of the outer loop.
  - The outer loop iterates n times for i going from 0 to $n-1$.
  - The cost of iteration i is i steps, and so the total cost of the outer loop is

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

- And so the total number of steps taken by all pairs is $\frac{n(n-1)}{2}$, which is $\Theta(n^2)$ QED

8.5 Analyzing Comprehensions and While Loops

Comprehensions

- Consider the following function

```
def square_all(numbers: List[int]) -> List[int]:
    """Return a new list containing the squares of the given numbers"""
    return [x ** 2 for x in numbers]
```

  - $Running\ time\ analysis$. We analyze it in the same way as a for loop
    - 1. We determine the number of steps required to evaluate the leftmost expression in the comprehension. In this case, evaluating x ** 2 takes 1 step
    - 2. The collection that acts as the source of the comprehension (i.e. numbers) determines how many times the leftmost expression is evaluated
  - Let n be the length of the input list numbers. The comprehension expression takes n steps
    - 1 step per element of numbers
  - The running time of square_all is n steps, which is $\Theta(n)$ QED
- The same analysis would hold in the above function if we had used a set or dictionary comprehension instead

While Loops

- Analysing the running time of code involving while loops follows the same principle as for loops
  - We calculate the sum of the different loop iterations (by multiplication/summation)
- **Ex.** Analyse the running time of the following function:

```
def my_sum_v2(numbers: List[int]) -> int:
    """Return the sum of the given numbers."""
    sum_so_far = 0
    i = 0
```

```
    while i < len(numbers):
        sum_so_far += numbers[i]
        i += 1

    return sum_so_far
```

- $Running\ time\ analysis$. Let n be the length of the input numbers
- We can divide up the function into 3 parts
  - 1. The cost of the assignment statements sum_so_far = 0 and i = 0 is constant time
  - 2. The while loop
    - Each iteration is constant time
    - There are n iterations, since i starts at 0 and increases by 1 until it reaches n
  - 3. The return statement takes constant time
- The total running time is $1 + n + 1 = n + 2$, which is $\Theta(n)$ QED

- **Ex.** Analyse the running time of the following function:

```
def my_sum_powers_of_two(numbers: List[int]) -> int:
    """Return the sum of the given numbers whose indexes are powers of 2.

    That is return numbers[1] + numbers[2] + numbers[4] + numbers[8] + …
    """
    sum_so_far = 0
    i = 1

    while i < len(numbers):
        sum_so_far += numbers[i]
        i *= 2

    return sum_so_far
```

- $Running\ time\ analysis$. Let n be the length of the input list numbers
- We count the initial assignment statements as 1 step, and the return statement as 1 step
- Each iteration takes constant time
- To determine the number of loop iterations, we follow these steps:
  - 1. Find a pattern for how i changes at each loop iteration, and a general formula for i_k, the value of i after k iterations

Iteration	Value of i
0	1

1	2
2	4
3	8
4	16

So we find that after k iterations, $i_k = 2^k$

- 2. We know the while loop continues while i < len(numbers)
  - i.e. the while loop continues *until* i >= len(numbers)

To find the number of iterations, we need to find the smallest value of k such that $i_k \geq n$, which makes the loop condition False

$$i_k \geq n$$
$$2^k \geq n$$
$$k \geq \log_2 n$$

So we need to find the smallest value of k such that $k \geq \log_2 n$, which is

$$\lceil \log_2 n \rceil$$

- The while loop iterates $\lceil \log_2 n \rceil$ times, with 1 step per iteration, for a total of $\lceil \log_2 n \rceil$ steps
- The function my_sum_powers_of_two has a running time of $1 + \lceil \log_2 n \rceil + 1 = \lceil \log_2 n \rceil + 2$, which is $\Theta(\log n)$ QED

A Trickier Example

- Example of a standard loop, with a twist in how the loop variable changes at each iteration

```
def twisty(n: int) -> int:
    """Return the number of iterations it takes for this special loop to stop
    for the given n.
    """
    iterations_so_far = 0
    x = n
    while x > 1:
        if x % 2 == 0:
            x = x / 2
        else:
            x = 2 * x – 2
        iterations_so_far += 1

    return iterations_so_far
```

- The loop variable x does not always get closer to the loop stopping condition
  - i.e. sometimes it increases

- We will perform an analysis based on *multiple iterations*
- $Claim.$ For any integer value of x greater than 2, after *two* iterations of the loop in twisty the value of x decreases by at least one.
- $Proof.$ Let x_0 be the value of variable x at some iteration of the loop, and assume $x_0 > 2$. Let x_1 be the value of x after one loop iteration, and x_2 be the value of x after two loop iterations. We want to prove that $x_2 \leq x_0 - 1$
    - We divide up this proof into four cases, based on the remainder of x_0 when dividing by 4
    - **Case 1:** Assume $4 \mid x_0$, i.e. $\exists k \in \mathbb{Z}, x_0 = 4k$
        - In this case, x_0 is even, so the if branch executes in the first loop iteration, and so $x_1 = \frac{x_0}{2} = 2k$. Then x_1 is also even, and so the if branch executes again: $x_2 = \frac{x_1}{2} = k$
        - So then $x_2 = \frac{1}{4}x_0 \leq x_0 - 1$ (since $x_0 \geq 4$), as required.
    - **Case 2:** Assume $4 \mid x_0 - 1$, i.e. $\exists k \in \mathbb{Z}, x_0 = 4k + 1$
        - In this case, x_0 is odd, so the else branch executes in the first loop iteration, and so $x_1 = 2x_0 - 2 = 8k$. Then x_1 is even, and so $x_2 = \frac{x_1}{2} = 4k$.
        - So then $x_2 = 4k = x_0 - 1$, as required.
    - **Case 3:** Assume $4 \mid x_0 - 2$, i.e. $\exists k \in \mathbb{Z}, x_0 = 4k + 2$
        - In this case, x_0 is even, so the if branch executes in the first loop iteration, and so $x_1 = \frac{x_0}{2} = 2k + 1$. The x_1 is odd, and so the else branch executes: $x_2 = 2x_1 - 2 = 4k$.
        - So then $x_2 = 4k \leq x_0 - 1$, as required.
    - **Case 4:** Assume $4 \mid x_0 - 3$, i.e. $\exists k \in \mathbb{Z}, x_0 = 4k + 3$
        - In this case, x_0 is odd, so the else branch executes in the first loop iteration, and so $x_1 = 2x_0 - 2 = 8k + 4$. Then x_1 is even, and so $x_2 = \frac{x_1}{2} = 4k + 2$.
        - So the $x_2 = 4k + 2 = x_0 - 1$, as required. QED
- $Running\ time\ analysis.$ (Analysis of twisty)
    - We count the variable initialization before the while loop as 1 step, and the return statement as 1 step
    - For the while loop:
        - The loop body takes 1 step
        - To count the number of loop iterations, we first observe that x starts at n and the loop terminates when x reaches 1 or less. The Claim tells us that after every 2 iterations, the value of x decreases by at least 1

        - So the after 2 iterations, $x \leq n - 1$; after 4 iterations, $x \leq n - 2$, and in general, after $2k$ iterations, $x \leq n - k$
        - This tells us that after $2(n - 1)$ loop iterations, $n \leq n - (n - 1) = 1$, and so the loop must stop.
    - This analysis tells us that the loop iterates *at most* $2(n - 1)$ times, and so takes *at most* $2(n - 1)$ steps
    - So the total running time of twisty is *at most* $1 + 2(n - 1) + 1 = 2n$ steps, which is $\mathcal{O}(n)$ QED
- We did not compute the exact number of steps the function twisty takes, only an *upper bound* on the number of steps
- We were only able to conclude a Big-O bound, and not a Theta bound
    - We don't know whether this bound is tight
- It is possible to prove something remarkable about what happens to the variable x after *three* iterations of the twisty loop
- $Claim$ (Improved). For any integer value of x greater than 2, let x_0 be the initial value of x and let x_3 be the value of x after *three* loop iterations. Then $\frac{1}{8}x_0 \leq x_3 \leq \frac{1}{2}x_0$
- The running time of twisty is both $\mathcal{O}(\log n)$ and $\Omega(\log n)$, and hence conclude that its running time is $\Theta(\log n)$

8.6 Analyzing Built-In Data Type Operations

Timing Operations

- Python provides a module called timeit that can tell us how long Python code takes to execute

```
>>> from timeit import timeit
>>> timeit('5 + 15', number=1000)
1.9799913324545578 4654e-05
```

    - The above call to timeit will perform the operation 5 + 15 one thousand times
    - The function returns the total time elapsed

How Python Lists are Stored in Memory

- All data used by a program are stored in blocks of computer memory, which are labeled by numbers called *memory addresses*
- For every Python list object, the references to its elements are stored in a *contiguous* block of memory
    - i.e. [8, 7, 100, -3]

400	601	(list index 0)
401	512	
402	699	
403	650	
404		
⋮		
512	7	
⋮		
601	8	
⋮		
650	-3	
⋮		
699	100	
⋮		

    - Each list elements are always stored consecutively
- This type of list implementation is called an *array-based list implementation*

Fast List Indexing

- Array-based list implementation makes list indexing fast
- List indexing is a *constant-time* operation
- True for both evaluating a list indexing expression or assigning to a list index

Mutating Contiguous Memory

- The references must always be stored in a contiguous block of memory
    - There can't be any "gaps"
- When a list element is deleted, all items after it have to be moved back one memory block to fill the gap
- Similarly, when a list element is inserted somewhere in the list, all items after it moves forward by one block
- When we remove the element at index i in the list, where $0 \leq i < n$, then $n - i - 1$ elements must be moved, and so the running time of this operation is $\Theta(n - i)$
- Inserting/deleting at the front of a Python list ($i = 0$) takes $\Theta(n)$ time
- Inserting/deleting at the back of a Python list ($i = n - 1$) is a constant time operation

Summary of List Operation Asymptotic Running Times (n is the list size)

Operation	Running time
List indexing (lst[i])	$\Theta(1)$

List index assignment (lst[i] = …)	$\Theta(1)$
List insertion at end (list.append(lst, …))	$\Theta(1)$
List deletion at end (list.pop(lst))	$\Theta(1)$
List insertion at index (list.insert(lst, i, …))	$\Theta(n-i)$
List deletion at index (list.pop(lst, i))	$\Theta(n-i)$

When Space Runs Out

- We assume that there will always be free memory blocks at the end of the list for the list to expand into

Running-Time Analysis with List Operations

- **Ex.** Analyse the running time of the following function

```
def squares(number: List[int]) -> int:
    """Return a list containing the square of the given numbers."""
    squares_so_far = 0

    for number in numbers:
        list.append(squares_so_far, number ** 2)

    return squares_so_far
```

  - $Running\ time\ analysis$. Let n be the length of the input list (i.e. numbers)
  - The assignment statement counts as 1 step
  - The for loop:
    - Takes n iterations
    - list.append takes *constant time*, and so the entire loop body counts as 1 step
    - This means the for loop takes $n \cdot 1 = n$ steps total
  - The return statement counts as 1 step
  - The total running time is $1 + n + 1 = n + 2$, which is $\Theta(n)$ QED
- **Ex.** Analyse the running time of the following function

```
def squares_reversed(numbers: List[int]) -> int:
    """Return a list containing the squares of the given numbers, in reverse
    order."""
    squares_so_far = 0

    for number in numbers:
        # Now, insert number ** 2 at the START of squares_so_far
        list.insert(squares_so_far, 0, number ** 2)
```

return squares_so_far

- $Running\ time\ analysis$. Let n be the length of the input list (i.e., numbers)
- The assignment statement counts as 1 step
- The for loop:
  - Takes n iterations
  - Inserting at the front of a Python list causes all of its current elements to be shifted over, taking time proportional to the size of the list. Therefore this call takes $\Theta(k)$ time, where k is the current length of squares_so_far
  - We know that squares_so_far starts as empty, and then increases in length by 1 at each iteration. So then k takes on the values 0, 1, 2, …, $n-1$, and we can calculate the total running time of the for loop using a summation:

$$\sum_{k=0}^{n-1} k = \frac{(n-1)n}{2}$$

- The return statement counts as 1 step
- The total running time is $1 + \frac{(n-1)n}{2} + 1 = n + 2$, which is $\Theta(n^2)$ QED

Sets and Dictionaries

- Both are implemented using a data structure called a *hash table*
  - Which allows *constant-time lookup*, *insertion*, and *removal* of elements (for a set) and key-value pairs (for a dictionary)
- Downside: not mutable

Data Classes

- Data classes store their instance attributes using a dictionary that maps attribute names to their corresponding values
- Data classes benefit from the constant-time dictionary operations above
- The two operations that we can perform on a dataclass instance: looking up an attribute value (i.e. david.age), and mutating the instance by assigning to an attribute (i.e. david.age = 99) both take constant time

Summary of Set, Dictionary, and Data Class Operations

Operation	Running time
Set/dict search (in)	$\Theta(1)$
set.add/set.remove	$\Theta(1)$
Dictionary key lookup (d[k])	$\Theta(1)$

Dictionary key assignment (d[k] = …)	$\Theta(1)$
Data class attribute access (obj.attr)	$\Theta(1)$
Data class attribute assignment (obj.attr = …)	$\Theta(1)$

Aggregation Functions

- sum, max, min have a *linear* running time ($\Theta(n)$), proportional to the size of the input collection
    - Each element of the collection must be processed in order to calculate each of these values
- len has a *constant* running time ($\Theta(1)$), independent of the size of input collection
    - The Python interpreter does *not* need to process each element of a collection when calculating the collection's size
    - Each of these collection data types stores a special attribute referring to the size of that collection
- any and all need to check every element of their input collection, but they can *short*-circuit (stopping before checking every element)
    - Similar to the logical or and and operators
    - Their running time isn't a fixed function of the input size, but rather a possible range of values, depending on whether this short-circuiting happens or not

8.7 Worst-Case Running Time Analysis

Intro

- Algorithms often depend on the actual value of the input, not just its size
- Consider the following function

```
def has_even(numbers: List[int]) -> bool:
    """Return whether numbers contain an even element."""
    for number in numbers:
        if number % 2 == 0:
            return True
    return False
```

    - Because this function returns as soon as it finds an even number in the list, its running time is not necessarily proportional to the length of the input list
- The running time of a function can vary even when the input size is fixed
    - $I_{has\_even,10}$ do *not* all have the same runtime
- Because our asymptotic notation is used to describe the growth rate of *functions*, we cannot use it to describe the growth of a whole range of values with respect to increasing input sizes

- We focus on the *maximum* of this range, which corresponds to the *slowest* the algorithm could run for a given input size
- $Definition.$ Let func be a program. We define the function $WC_{func} : \mathbb{N} \to \mathbb{N}$, called the *worst-case running time function of* func, as follows:

$$WC_{func}(n) = \max\{\text{running time of executing } func(x) \mid x \in I_{func,n}\}$$

- WC_{func} is a function, not a constant number: it returns the maximum possible running time for an input of size n, for every natural number n
  - And so we can use asymptotic notation to describe it
- The goal of a *worst-case runtime analysis* for func is to find an elementary function f such that $WC_{func} \in \Theta(f)$
- We take a two-pronged approach: proving matching *upper* and *lower bounds* on the worst-case running time of our algorithm

Upper Bounds on the Worst-Case Runtime

- $Definition.$ Let func be a program, and WC_{func} its worst-case runtime function. We say that a function $f : \mathbb{N} \to R^{\geq 0}$ is an **upper bound on the worst-case runtime** when $WC_{func} \in \mathcal{O}(f)$
- Suppose we use absolute dominance rather than Big-O, expanding the phrase "WC_{func} is absolutely dominated by f":

$$\forall n \in \mathbb{N}, WC_{func}(n) \leq f(n)$$
$$\iff \forall n \in \mathbb{N}, \max\{\text{running time of executing } func(x) \mid x \in I_{func,n}\} \leq f(n)$$
$$\iff \forall n \in \mathbb{N}, \forall x \in I_{func,n}, \text{ running time of executing } func(x) \leq f(n)$$

  - An upper bound on the worst-case runtime is equivalent to an upper bound on the runtimes of *all* inputs
- Translation of $WC_{func} \in \mathcal{O}(f)$:

$$\exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0$$
$$\implies \left(\forall x \in I_{func,n}, \text{ running time of executing } func(x) \leq c \cdot f(n)\right)$$

- To approach an analysis of an upper bound on the worst-case, we typically find a function g such that WC_{func} is absolutely dominated by g, and then find a simple function f such that $g \in \mathcal{O}(f)$
- **Ex.** Find an asymptotic upper bound on the worst-case running time of has_even
  - $Discussion.$ The intuitive translation using absolute dominance is usually enough for an upper bound analysis
  - $Running\ time\ analysis.$ (Upper bound on worst-case)
    - Let $n \in \mathbb{N}$, let numbers be an *arbitrary* list of length n
    - The loop iterates *at most* n times. Each loop iteration counts as a single step, so the loop takes *at most* $n \cdot 1 = n$ steps in total.

        - The return False statement (if it is executed) counts as 1 basic operation.
        - Therefore the running time is *at most* $n + 1$, and $n + 1 \in \mathcal{O}(n)$. So we can conclude that the worst-case running time of has_even is $\mathcal{O}(n)$. QED
- Because we calculated an upper bound rather than an exact number of steps, we can only conclude a Big-O
    - We don't yet know that this upper bound is tight

Lower Bounds on the Worst-Case Runtime

- $Definition.$ Let func be a program, and WC_{func} is the worst-case runtime function. We say that a function $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ is a **lower bound on the worst-case runtime** when $WC_{func} \in \Omega(f)$
- Using absolute dominance,

$$\forall n \in N, WC_{func}(n) \geq f(n)$$
$$\Leftrightarrow \forall n \in \mathbb{N}, \max\{\text{running time of executing } func(x) \mid x \in I_{func,n}\} \geq f(n)$$
$$\Leftrightarrow \forall n \in N, \exists x \in I_{func,n}, \text{ running time of executing } func(x) \geq f(n)$$

- Our goal is to find an *input family* whose runtime is asymptotically larger than our target lower bound
    - *Input family* – a *set* of inputs, one per input size n
- In has_even we want to prove that the worst-case running time is $\Omega(n)$ to match the $\mathcal{O}(n)$ upper bound
- **Ex.** Find an asymptotic lower bound on the worst-case running time of has_even
    - $Running\ time\ analysis.$ (Lower bound on worst-case)
        - Let $n \in \mathbb{N}$. Let numbers be the list of length n consisting of all 1's.
        - In this case, the if condition in the loop is always false, so the loop never stops early. Therefore it iterates exactly n times, with each iteration taking 1 step.
        - Finally, the return False statement executes, which is one step. So the total number of steps for this input is $n + 1$, which is $\Omega(n)$ QED

Putting It All Together

- We can combine our upper and lower bounds on WC_{has_even} to obtain a tight asymptotic bound
- **Ex.** Find a *tight* bound on the worst-case running time of has_even
    - $Running\ time\ analysis.$ Since we've proved that WC_{has_even} is $\mathcal{O}(n)$ and $\Omega(n)$, it is $\Theta(n)$. QED
- To obtain a tight bound on the worst-case running time of a function, we need to do 2 things:

- Use the properties of the code to obtain an *asymptotic upper bound* on the worst-case running time. We would say something like $WC_{func} \in \mathcal{O}(f)$
- Find a family of inputs whose running time is $\Omega(f)$. This will prove that $WC_{func} \in \Omega(f)$
- After showing that $WC_{func} \in \mathcal{O}(f)$ and $WC_{func} \in \Omega(f)$, we can conclude that $WC_f \in \Theta(f)$

A Note About Best-Case Runtime

- It is possible to define one
- Would not be useful to know

Early Returning in Python Built-Ins

- The worst-case running time of the in operation for lists is $\Theta(n)$, where n is the length of the list
- any and all have a worst-case running time of $\Theta(n)$, where n is the size of th input collection

Any, All, and Comprehensions

- The following two lines are slow despite the fact that the function could early-returned on the first element

```
>>> any([x == 0 for x in range(0, 1000000)])
>>> any({x == 0 for x in range(0, 1000000)})
```

- The *full comprehension* is evaluated before any is called
- We can write the comprehension expression in the function call without any surrounding square brackets or curly braces

```
>>> any(x == 0 for x in range(0, 1000000))
```

- The above example is called a *generator comprehension*, and is used to produce a special Python collection data type called a *generator*
- Generator comprehensions do not evaluate their elements all at once, but instead only when they are needed by the function being called
- In the previous example, only the x = 0 value from the generator comprehension gets evaluated

Don't Assume Bounds are Tight!

- **Ex.** We say that a string is a **palindrome** when it can be read the same forwards and backwards (i.e. 'abba', 'racecar', 'z'). We say that a string s_1 is a **prefix** of another string s_2 when s_1 is a substring of s_2 that starts at index 0 of s_2 (i.e. 'abc' is a prefix of 'abcde').

- The algorithm below takes a non-empty string as input, and returns the length of the longest prefix of that string that is a palindrome.
    - For example, the string "attack" has 2 non-empty prefixes that are palindromes, "a" and "atta", and so our algorithm will return 4

```
def palindrome_prefix(s: str) -> int:
    n = len(s)
    for prefix_length in range(n, 0, -1):  # goes from n down to 1
        # Check whether s[0:prefix_length] is a palindrome
        is_palindrome = all(s[i] == s[prefix_length – 1 – i] for i in range(0,
        prefix_length)

        # If a palindrome prefix is found, return the current length.
        if is_palindrome:
            return prefix_length
```

    - The for loop iterable is range(n, 0, -1)
        - The third argument -1 causes the loop variable to *start* at n and decrease by 1 at each iteration
            - The loop is checking the possible prefixes starting with the longest prefix (length n) and working its way to the shortest prefix
    - The call to all checks pairs of characters starting at either end of the current prefix
        - It uses a *generator comprehension* so that it can stop early as soon as it encounters a mismatch (i.e. when s[i] != s[prefix_length – 1 – i])
    - The algorithm is guaranteed to find a palindrome prefix despite the only return statement is inside the for loop, since the first letter of s by itself is a palindrome
- It is not too hard to show that the worst-case runtime of this function is $\mathcal{O}(n^2)$, but it is hard to show that the worst-case runtime is $\Omega(n^2)$
    - The call to all can stop as soon as the algorithm detects that a prefix is *not* a palindrome
    - The return statement occurs when the algorithm has determined that a prefix *is* a palindrome
- We can consider two extreme input families
    - The entire string s is a palindrome of length n. In this case, in the first iteration of the loop, the entire string is checked. The all call checks all pairs of characters, but this means that is_palindrome = True, and the loop returns during its very firsts iterations
        - Since the all call takes n steps, this input family takes $\Theta(n)$ time to run

    - The entire string s consists of n different letters. In this case, the only palindrome prefix is just the first letter of s itself. This means that the loop will run for all n iterations, only returning in its last iteration (i.e. when prefix_length == 1).
        - However, the all call will always stop after just 1 step, since it starts by comparing the first letter of s with another letter, which is guaranteed to be different by our choice of input family.
        - Leads to a $\Theta(n)$ running time
- We want to choose an input family that *doesn't* contain a long palindrome (so the loop runs for many iterations), but whose prefixes are close to being palindromes (so the all call checks many pairs of letters)
- Let $n \in \mathbb{Z}^+$. We define the input s_n as follows:
    - $s_n\left[\left\lfloor\frac{n}{2}\right\rfloor\right] = b$
    - Every other character in s_n is equal to a

  For example, $s_4 = aaba$ and $s_{11} = aaaaaabaaaa$
    - s_n is very close to being a palindrome
    - The longest palindrome of s_n has length roughly $\frac{n}{2}$
        - The loop iterates roughly $\frac{n}{2}$ times
    - The "outer" characters of each prefix of s_n containing more than $\frac{n}{2}$ characters are all the same
        - The all call checks many pairs to find the mismatch between a and b
    - This input family has an $\Omega(n^2)$ runtime

8.8 Testing Functions IV: Efficiency

An Efficiency Test

- Consider the following example

```
from math import floor, sqrt
from timeit import timeit

def is_prime(p: int) -> bool:
    """Return whether p is prime."""
    possible_divisors = range(2, floor(sqrt(p)) + 1)
    return p > 1 and all(not p % d == 0 for d in possible_divisors))

def test_is_prime_performance() -> None:
    """Test the efficiency of is_prime."""
```

```
numbers_to_test = range(2, 1000)
for number in numbers_to_test:
    time = timeit(f'is_prime({number})', number=100,
    globals=globals())
    assert time < 0.001, 'Failed performance constraint of 0.001s.'
```

- Where did number=100 come from?
  - Since there are many external factors that can impact the results, several samples of an experiment (i.e. measurements of time) need to be taken
- Where did 0.001 seconds come from?
  - This number is arbitrary as computer systems are different from one another
  - This part may be tuned over time in the testing suite
  - May help identify the minimum hardware requirements for running a piece of software
- It is challenging to come up with the actual parameters
  - i.e. number of function calls, inputs to the function, total acceptable runtime
- When a code change causes an efficiency test to fail, the programmers can decide whether to the change efficiency constraint or explore alternative code changes
  - Without efficiency tests in place, the change in performance might not have been found until it impacted a real user of the software

9.1 An Introduction to Abstraction

Intro

- We can think of abstraction as allowing for the separation of two groups of people with different goals:
  - The *creators* of an entity
    - Responsible for designing, building, and implementing an entity
  - The *users* (or *clients*) of that entity
    - Responsible for using it
- The *interface* of an entity is the boundary between creator and user
  - *Interface* – the set of rules (implicit or explicit) governing how users can interact with that entity
    - The *public* side of an entity
    - The part of the creator's work that everyone can interact it

Abstraction in Computer Science

- We are *users* of the Python programming language, which provides an interface that hides the details of our computer hardware, processor instructions, memory, storage, and graphics
- We are *users* of built-in Python functions, data types, and modules
- We are *creators* of new Python functions, data types and modules
- For a *function*, its interface is its header and docstring
    - These specify how to call the function, the arguments it expects, and what the function does
    - The function body is not part of the interface
- For a *data class*, its interface is the name of the data class and the names and types of its attributes, and the class docstring
    - *Every* part of what we write to define a new data class is part of its interface
- For a Python *module*, the interface is the collection of interfaces of the functions and data types defined in that module, plus any additional documentation in the module
- Every time we write a proof, we act as a creator of knowledge, providing airtight evidence that a statement is True
- Every time we use an "external statement" in a proof (i.e. Quotient-Remainder Theorem, Fermat's Little Theorem), we are acting as *users* of these statements, and do not worry about how they have been proved

Interfaces are Contracts

- Every interface is a contract between creator and user
    - Creators have the responsibility to make the interface easy and intuitive for users
- When we act as the creators of a function or module, we are free to modify their implementations in any way we wish, as long as we do not change the public interface

9.2 Defining Our Own Data Types, Part 3

What If We Just Remove the @dataclass?

- Recall the Person data class example

```
# @dataclass
class Person:
    """A custom data type that represents data for a person."""
    given_name: str
    family_name: str
    age: int
    address: str
```

- If we remove the @dataclass decorator from our class definition (it is commented out), we get an unexpected consequence

```
>>> david = Person('David', 'Liu', 100, '40 St. George Street')
TypeError: Person() takes no arguments
```

- We can create an instance of the Person class passing in zero arguments

```
>>> david = Person()
>>> type(david)
<class 'Person'>
```

Defining an Initializer

- A Person object has been created, but it has no attributes
- We need to define a new method for Person called the *initializer*
  - The initializer method of a class is called when an instance of the class is created in Python
  - Purpose: to initialize all of the instance attributes for the new object
  - Python always use the name __init__ for the initializer method
- When we use the @dataclass decorator the Python interpreter automatically creates an initializer method for the class, like what is shown below:

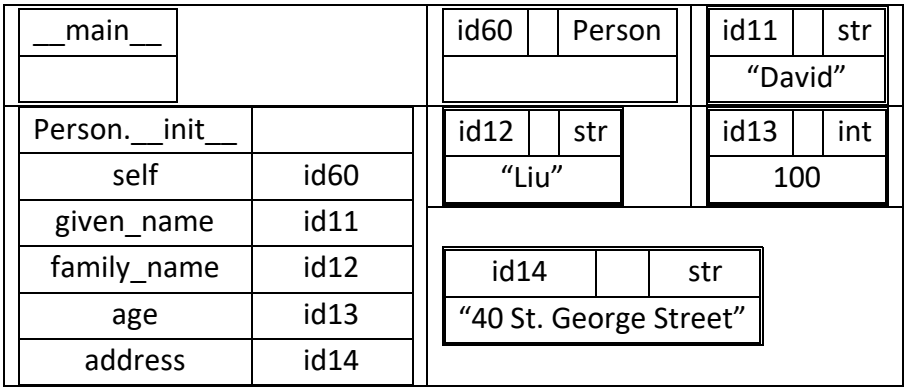
```
class Person:
    """A custom data type that represents data for a person."""
    given_name: str
    family_name: str
    age: int
    address: str

    def __init__(self, given_name: str, family_name: str, age: int, address: str)
    -> None:
        """Initialize a new Person object."""
        self.given_name = given_name
        self.family_name = family_name
        self.age = age
        self.address = address
```

- This method is *indented* so that it is inside the body of the class Person definition
- Every initializer has a first parameter that refers to the instance that has just been created and is to be initialized
  - By convention, we always call it self
  - We could have written self: Person, but it is redundant because the type for self should *always* be the class that the initializer belongs to

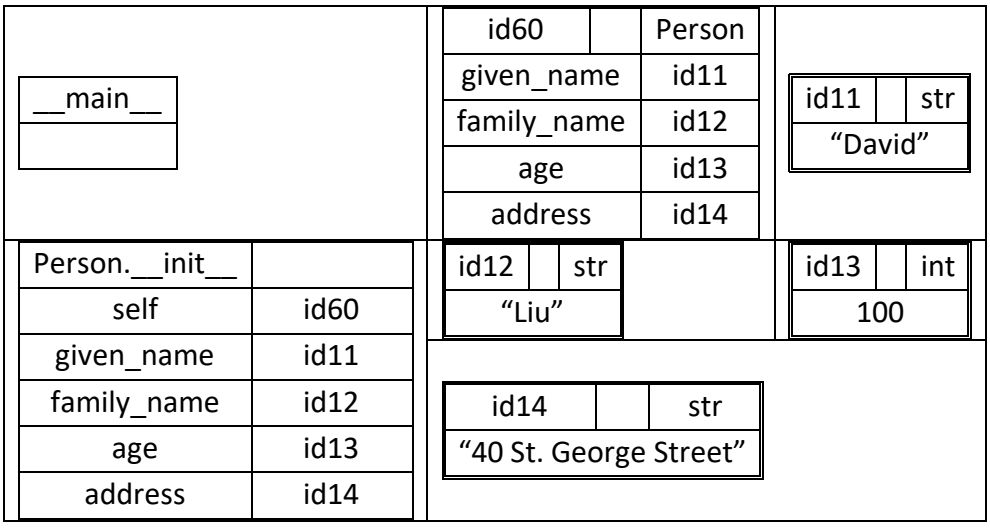
- We use the initializer by calling the data class as usual:

  >>> david – Person('david', 'Liu', 100, '40 St. George Street')

  - The initializer is called automatically
  - We never have to pass a value for self
    - Python automatically sets it to the instance that is to be initialized
- Memory at the beginning of the initializer:



- The initializer's job is to create and initialize the instance attributes
  - To do this, we use one assignment statement per instance attribute
    - Uses the same dot notation syntax for assigning to instance attributes
  - given_name is a *parameter* of the initialize
  - self.given_name is an *instance attribute*
- Memory immediately before the initializer returns:



What Really Happens When We Create a New Object

- Person doesn't just cause __init__ to be called, instead, it does three things:
  - Create a new Person object behind the scenes
  - Call __init__ with the new object passed to the parameter self, along with the other arguments

    - Return the new object. This step is where the object is returned, not directly from the call to __init__ in step 2
- __init__ is a *helper function* in the object creation process
    - Its task is only to initialize attributes for an object
    - Python handles both creating the object beforehand, and returning the new object after __init__ has been called

Methods as Part of a Data Type Interface

- We can define *methods* for a data type, which become part of the interface of that data type
- When we define a class with methods, those methods are *always* bundled with the class, and so any instance of the class can use those methods, without needing to import them separately
- One example of a method definition is the initializer, __init__
- Any function that operates on an instance of a class can be converted into a method by doing the following:
    - Indent the function so that it is part of the clss body, underneath the instance attributes
    - Ensure that the first parameter of the function is an instance of the class, and name this parameter self
- For example, suppose we had the following function to increase a person's age:

```
def increase_age(person: Person, years: int) -> None:
    """Add the given number of years to the given person's age."""
    person.age = person.age + years
```

- We can turn increase_age into a Person method as follows:

```
class Person:
    """A custom data type that represents data for a person."""
    given_name: str
    family_name: str
    age: int
    address: str

    def __init__(self, given_name: str, family_name: str, age: int, address: str)
    -> None:
        """Initialize a new Person object."""
        self.given_name = given_name
        self.family_name = family_name
        self.age = age
```

```
        self.address = address

    def increase_age(self, years: int) -> None:
        """Add the given number of years to this person's age"""

        self.age = self.age + years
```

- We use parameter self (without a type annotation) to access instance attributes, just like what we did in the initializer

Shortcut Syntax for Method Calls

- We can call increase_age like this:

```
>>> Person.increase_age(david, 10)
```

- The alternate form for calling the increase_age method is to use dot notation *with the* Person *instance directly*:

```
>>> david.increase_age(10)
```

- When we call david.increase_age(10), the Python interpreter does the following:
  - It looks up the class of david, which is Person
  - It looks up the increase_age method of the Person class
  - It looks up the increase_age on david and 10
    - The interpreter *automatically* passes the value to the left of the dot (i.e. david) as the method's first parameter self
- This works for all built-in data types as well
  - list.append(lst, 10) can be written as lst.append(10)
  - str.lower(s) can be written as s.lower()
  - obj.method(x1, x2, …, xn) is equivalent to type(obj).method(obj, x1, x2, …, xn)
- The "object dot notation" style (i.e. david.increase_age) is the more common one in Python programming
  - It matches other languages with an *object-oriented* style of programming, where the object being operated own is of central importance
    - david.increase_age(10) implies that david is the most important object in the code expression
  - Only the "object dot notation" style supports *inheritance*

9.3 Data Types, Abstract and Concrete

Intro

- *Concrete data types* – synonymous to *Python class*
  - Have concrete implementations in Python code

- *Abstract data type (ADT)* – defines an entity that stores some kind of data and the operations that can be performed on it
    - Language-independent
    - Pure interface concerned only with the *what* (i.e. what data is stored, what we can do with the data) and not the *how* (i.e. how a computer actually stores this data or implements these operations)

Familiar Abstract Data Types

- *Set*
    - Data: a collection of unique elements
    - Operations: get size, insert a value (without introducing duplicates), remove a specified value, check membership in the set
- *List*
    - Data: an ordered sequence of elements (which may or may not be unique)
    - Operations: get size, access element by index, insert a value at a given index, remove a value at a given index
- *Iterable*
    - Data: a collection of values (may or may not be unique)
    - Operations: iterate through the elements of the collection one at a time
- ADT's are abstract enough to transcend any individual program or even programming languages

Abstract vs. Concrete Data Types

- dict is not an abstract data type, but it is an obvious implementation of the Mapping ADT
- There is *not* a one-to-one correspondence between abstract data types and concrete data types, in Python or any other programming language
- A single abstract data type can be implemented by many different concrete data types
    - {0: 'hello', 1: 42, 2: 'goodbye'} # A Map using a Python dict
    - [(0, 'hello'), (1, 42), (2, 'goodbye')] # A Map using a Python list
- Every concrete data type can be used to implement multiple ADTs
- i.e. it is possible to implement a Map using list, but this choice is worse than using dict
    - Because of *efficiency*

9.4 Stacks

The Stack ADT

- A stack contains 0 or more items

- When we add an item, it goes “on the top” of the stack
    - “Pushing” onto the stack
- When we remove an item, it is removed from the top
    - “Popping” from the stack
- *Last-In-First-Out (LIFO)* behaviour
- *Stack*
    - Data: a collection of items
    - Operations: determine whether the stack is empty, add an item (*push*), remove the most recently-added item (*pop*)

```
class Stack
    """A last-in-first-out (LIFO) stack of items.
    Stores data in last-in, first-out order. When removing an item from the
    stack, the most recently-added item is the one that is removed.
    Sample Usage:
    >>> s = Stack()
    >>> s.is_empty()
    True
    >>> s.push('hello')
    >>> s.is_empty()
    False
    >>> s.push('goodbye')
    >>> s.pop()
    'goodbye'
    """
    def __init__(self) -> None:
        """Initialize a new empty stack."""

    def is_empty(self) -> bool:
        """Return whether this stack contains no items."""

    def push(self, item: Any) -> None:
        """Add a new element to the top of this stack."""

    def pop(self) -> Any:
        """Remove and return the element at the top of this stack.

        Preconditions:
            - not self.is_empty()
```

"""

Applications of Stacks

- "Undo" feature
    - We want to undo the *most recent* action

Implementing the Stack ADT Using Lists

- We can implement the Stack ADT using list
- We've chosen to use the *end* of the list to represent the top of the stack

```
class Stack1
    """..
    Instance Attributes:
        - items: The items stored in the stack. the end of the list
        represents the top of the stack.
    ..."""
    items: list

    def __init__(self) -> None:
        """..."""
        self.items = []

    def is_empty(self) -> bool:
        """..."""
        return self.items == []

    def push(self, item: Any) -> None:
        """..."""
        self.items.append(item)

    def pop(self) -> Any:
        """..."""
        return self.items.pop()
```

Attributes and the Class Interface

- Users can also access the items instance attribute
- To make an instance attribute that *isn't* part of a class' interface, we prefix its name with an underscore _
- *Private instance attributes* – attributes whose names begin with an underscore

- *Public instance attributes* – attributes whose names do not begin with an underscore
- All public instance attributes are part of the interface, all private ones aren't

```
class Stack1:
    """..."""
    # Private Instance Attributes:
    #     - _items: The items stored in the stack. The end of the list
    #     represents the top of the stack.
    _items: list

    def __init__(self) -> None:
        """..."""
        self._items = []

    def is_empty(self) -> bool:
        """..."""
        return self._items == []

    def push(self, item: Any) -> None:
        """..."""
        self._items.append(item)

    def pop(self) -> Any:
        """..."""
        return self._items.pop()
```

- There is no mention of the attribute _items when we call help on our class

Warning: Private Attributes Can Be Accessed!

- Private instance attributes can still be accessed from outside the class
- The Python programming language prefers *flexibility* over *restriction* when it comes to accessing attributes
- By making an instance attribute private, we are communicating that client code should *not* access this attribute
    - We reduce the cognitive load on the client
    - We also give flexibility to the designer of the class to change or even remove a private attribute if they want to update their implementation of the class, without affecting the class' public interface

Analyzing Efficiency

- We could have implemented Stack1 using the front of _items to represent the top of the stack

```
class Stack2:
    # Duplicated code from Stack1 omitted. Only push and pop are different.

    def push(self, item: Any) -> None:
        """..."""
        self._items.insert(0, item)

    def pop(self0 -> Any:
        """..."""
        return self._items.pop(0)
```

- Key difference between Stack1 and Stack2: *efficiency*
  - Running time of Stack1.push: $\Theta(1)$
  - Running time of Stack2.push: $\Theta(n)$
  - Running time of Stack1.pop: $\Theta(1)$
  - Running time of Stack2.pop: $\Theta(n)$

9.5 Exceptions As a Part of the Public Interface

Letting an Error Happen

- Consider this version of Stack.pop, which removes the precondition but keeps the same implementation

```
def pop(self) -> Any:
    """..."""
    return self._items.pop()
```

  - When we call pop on an empty stack, we encounter an IndexError
  - From the perspective of the client code, it is bad that the exception report refers to a list (IndexError: pop from empty list) and a private attribute (self._items) that the client code should have no knowledge of

Custom Exceptions

- A better solution is to raise a custom exception that is descriptive, yet does not reveal any implementation details
- We can define our own type of error by defining a new class

```
class EmptyStackError(Exception):
    """Exception raised when calling pop on an empty stack."""
```

- To use EmptyStackError in our pop method,

```
def pop(self) -> Any:
    """Remove and return the element at the top of this stack.
    Raise a EmptyStackError if this stack is empty.
    """
    if self.is_empty():
        raise EmptyStackError
    else:
        return self._items.pop()
```

- The exception is now part of the *public interface* because the docstring names both the type of exception and the scenario that will cause that exception to be raised
- The Python keyword raise will raise an exception
- When we call pop on an empty stack, it will display EmptyStackError rather than mentioning any implementation details

Custom Exception Messages

- To provide a custom exception message, we can define a new special method with the name __str__ in our exception class:

```
class EmptyStackError(Exception):
    """..."""
    def __str__(self) -> str:
        """Return a string representation of this error."""
        return 'pop may not be called on an empty stack'
```

- Now it will display EmptyStackError: pop may not be called on an empty stack

Testing Exceptions

- We cannot simply call pop on an empty stack and check the return value or the state of stack after pop returns
    - Raising an error interrupts the regular control flow of a Python program
- The pytest module allows us to write tests that expects an exception to occur using a function pytest.raises together with the with keyword

```
# Assuming our stack implementation is contained in a file stack.py.
from stack import Stack, EmptyStackError
import pytest

def test_empty_stack_error():
    """Test that popping from an empty stack raises an exception."""
    s = Stack()
```

```
with pytest.raises(EmptyStackError):
    s.pop()
```

- The test *passes* when that exception is raised, and *fails* when that exception is not raised
  - Also fails when a different exception is raised

Handling Exceptions

- Python provides the *try-except* statement to execute a block of code and handle3 a case where one or more pre-specified exceptions are raised in that block
  - The simplest form of a try-except statement:

```
try:
    <statement>
    …
except <ExceptionClass>:
    <statement>
    …
```

- When a try-except statement is executed:
  - The block of code indented within the try is executed
  - If no exception occurs when executing this block, the except part is skipped, and the Python interpreter continues to the next statement after the try-except
  - If an exception occurs when executing this block:
    - If the exception has type <ExceptionClass>, the block under the except is executed, and then after that the Python interpreter continues executing the next statement after the try-except
      - In this case the problem does *not* immediately half
    - If the exception is a different type, this does stop the normal program execution
- Try-except statements shield users from seeing errors that they should never see, and allows the rest of the program to continue
- Example: a function that takes a stack and returns the second item from the top of the stack

```
def second_from_top(s: Stack) -> Optional[str]:
    """Return the item that is second from the top of s.
    If there is no such item in the Stack, returns None.
    """
    try:
        hold1 = s.pop()
    except EmptyStackError:
```

```
            # In this case, s is empty. We can return None.
            return None

        try:
            hold2 = s.pop()
        except EmptyStackError:
            # In this case, s had only 1 element
            # We restore s to its original state and return None
            s.push(hold1)
            return None

        # If we reach this point, both of the previous s.pop() calls succeeded.
        # In this case, we restore s to its original state and return the second
        item.
        s.push(hold2)
        s.push(hold1)

        return hold2
```

9.6 Queues

The Queue ADT

- Unlike a stack, items come out of a queue in the order in which they entered
  - *First-In-First-Out (FIFO)*
- *Queue*
  - Data: a collection of items
  - Operations: determine whether the queue is empty, add an item (*enqueue*), remove the least recently-added item (*dequeue*)

```
class Queue:
    """A first-in-first-out (FIFO) queue of items.
    Stores data in a first-in, first-out order. When removing an item from the
    queue, the most recently-added item is the one that is removed
    >>> q = Queue()
    >>> q.is_empty()
    True
    >>> q.enqueue('hello')
    >>> q.is_empty()
    False
```

```
>>> q.enqueue('goodbye')
>>> q.dequeue()
'hello'
>>> q.dequeue()
'goodbye'
>>> q.is_empty()
True

def __init__(self) -> None:
    """Initialize a new empty queue."""

def is_empty(self) -> bool:
    """Return whether this queue contains no items."""

def enqueue(self, item: Any) -> None:
    """Add <item> to the back of this queue."""

def dequeue(self) -> Any:
    """Remove and return the item at the front of this queue.

    Raise an EmptyQueueError if this queue is empty.
    """

class EmptyQueueError(Exception):
    """Exception raised when calling dequeue on an empty queue."""

    def __str__(self) -> str:
        """Return a string representation of this error."""
        return 'dequeue may not be called on an empty queue'
```

List-Based Implementation of the Queue ADT

- We have decided that the beginning of the list (i.e. index 0) is the front of the queue

```
class Queue:
    """..."""
    # Private Instance Attributes:
    #     - _items: The items stored in this queue. The front of the list
    #     represents the front of the queue.
    _items: list
```

```
def __init__(self) -> None:
    """..."""
    self._items = []

def is_empty(self) -> bool:
    """..."""
    return self._items == []

def enqueue(self, item: Any) -> None:
    """..."""
    self._items.append(item)

def dequeue(self) -> Optional[Any]:
    """..."""
    if self.is_empty():
        raise EmptyQueueerror
    else:
        return self._items.pop(0)
```

Implementation Efficiency

- Our Queue.enqueue calls list.append, which takes constant time
- Our Queue.dequeue calls self._items.pop(0), which takes $\Theta(n)$ time
- If we change things around so that the front of the queue is the end of the list (rather than the beginning), we simply swap these running times
- Using an array-based list, we can *either* have an efficient enqueue or an efficient dequeue operation

9.7 Priority Queues

The Priority Queue ADT

- Items are removed from a Priority Queue in order of their priority
- *Priority Queue*
    - Data: a collection of items and their priorities
    - Operations: determine whether the priority queue is empty, add an item with a priority (*enqueue*), remove the highest priority item (*dequeue*)
- We will represent priorities as integers, with larger integers representing higher priorities

```
class PriorityQueue
    """A collection items that are be removed in priority order.
    When removing an item from the queue, the highest-priority item is the
    one that is removed.
    >>> pq = PriorityQueue()
    >>> pq.is_empty()
    True
    >>> pq.enqueue(1, 'hello')
    >>> pq.is_empty()
    False
    >>> pq.enqueue(5, 'goodbye')
    >>> pq.enqueue(2, 'hi')
    >>> pq.dequeue()
    'goodbye'
    """

    def __init__(self) -> None:
        """Initialize a new and empty priority queue."""

    def is_empty(self) -> bool:
        """Return whether this priority queue contains no items."""

    def enqueue(self, priority: int, item: Any) -> None:
        """Add the given item with the given priority to this priority
        queue.
        """

    def dequeue(self) -> Any:
        """Remove and return the item with the highest priority.
        Raise an EmptyPriorityQueueError when the priority queue is
        empty.
        """

class EmptyPriorityQueueError(Exception):
    """Exception raised when calling dequeue on an empty priority queue."""

    def __str__(self) -> str:
        """Return a string representation of this error."""
```

return 'You called dequeue on an empty priority queue.'

List-Based Implementation of the Priority Queue ADT

- Our implementation idea here is to use a private attribute that is a *list of tuples*, where each tuple is a (priority, item) pair
- Our list will also be *sorted* with respect to priority (breaking ties by insertion order), so that the *last* element in the list is always the next item to be removed from the priority queue

```
class PriorityQueue:
    """..."""
    # Private Instance Attributes:
    #     - _items: a list of the items in this priority queue
    _items: List[Tuple[int, Any]]

    def __init__(self) -> None:
        """..."""
        self._items = []

    def if_empty(self) -> bool:
        """..."""
        return self._items == []

    def dequeue(self) -> Any:
        """..."""
        if self.is_empty():
            raise EmptyPriorityQueueError
        else:
            _priority, item = self._items.pop()
            return item
```

  - For PriorityQueue.enqueue, we can insert the new priority and item into the list, and then sort the list by priority, however, this is inefficient
  - Our enqueue implementation will search for the right index in the list to add the new item

```
def enqueue(self, priority: int, item: Any) -> None:
    """..."""
    i = 0
    while i < len(self._items) and self._items[i][0] < priority:
        # Loop invariant: all items in self._items[0:i]
```

```
                # have a lower priority than <priority>
                i = i + 1

            self._items.insert(i, (priority, item))
```

  - It should be < rather than <= because if there already exist items with the same priority, the new one should be the least-prioritized (more in front than other items with the same priority but behind items with lower priority)
- Analysis
  - The while loop takes *at most* n iterations, since i starts at 0 and increases by 1 at each iteration, and the loop must stop when i reaches n (if it hasn't stopped earlier)
    - Since each loop iteration takes 1 step, in total the while loop takes at most n steps
  - list.insert takes at most n steps, where n is the length of the list being inserted into
  - Adding up these two quantities, the total running time of this algorithm is at most $n + n = 2n$ steps, which is $\mathcal{O}(n)$
- We can incorporate the value of variable i in our calculation. Let I be the value of variable i *after* the loop finishes. Then:
  - We now know that the while loop takes *exactly* I iterations, for a total of I steps (1 step per iterations)
  - We know that calling list.insert on a list of length n to insert an item at index I takes $n - I$ steps
  - So the total running time is actually $I + (n - I) = n$ steps, which is $\Theta(n)$
- We've shown that *every* call to this implementation of PriorityQueue.enqueue will take $\Theta(n)$ time, regardless of the priority being inserted.

Using an Unsorted List

- If we used an unsorted list of tuples instead, we would have $\Theta(1)$ enqueue operations, simply by appending a new (priority, item) tuple to the end of self._items
- However, we must search for the highest priority item in a list of unsorted items, which would take $\Theta(n)$ time

Looking Ahead: Heaps

- *Heap* – a data structure commonly used to implement the Priority Queue ADT in practice
- We can use this data structure to implement both PriorityQueue.enqueue and PriorityQueue.dequeue with a worst-case running time of $\Theta(\log n)$

9.8 Defining a Shared Public Interface with Inheritance

Intro

- Recall that the Stack ADT can be implemented using a Python list in 2 ways:
    - Storing the top of the stack at the end of the list (Stack1)
    - Storing the top of the stack at the front of the list (Stack2)
- They share the same *public interface* of the Stack ADT

The Stack Abstract Class

- Defining a Stack class that consists only of the *public interface* of the Stack ADT

```
class Stack:
    """A last-in-first-out (LIFO) stack of items.

    This is an abstract class. Only subclasses should be initiated.
    """

    def is_empty(self) -> bool:
        """..."""
        raise NotImplementedError

    def push(self, item: Any) -> None:
        """..."""
        raise NotImplementedError

    def pop(self) -> Any:
        """..."""
        raise NotImplementedError

class EmptyStackError(Exception):
    """Exception raised when calling pop on an empty stack."""
```

- In Python, we mark a method as unimplemented by having its body raise a special exception, NotImplementedError
    - We say that a method is *abstract* when it is not implemented and raises this error
    - We say that a *class* is *abstract* when at least one of its methods is abstract (i.e. not implemented)

    - A *concrete class* is a class that is not abstract
- The Stack class we have defined is a direct translation of the Stack ADT: an *interface* that describes the methods that a concrete class that wants to implement the Stack ADT *must define*

Inheriting the Stack Abstract Class

- Earlier, we defined Stack1 and Stack2, however, the code did not indicate that the types were related in any way
- With the abstract class Stack, we can indicate this relationship in the code through *inheritance*

```
class Stack1(Stack):
    """..."""

class Stack2(Stack):
    """..."""
```

    - The syntax (Stack) indicates that Stack1 and Stack2 inherit from Stack
    - Stack: base class, superclass, parent class
    - Stack1, Stack2: subclass, child class, derived class
- When one class in Python inherits from another,
    - The Python interpreter treats every instance of the subclass as an instance of the superclass as well

```
>>> s1 = Stack()
>>> isinstance(s1, Stack1)
True
>>> isinstance(s1, Stack)
True
>>> isinstance(s1, Stack2)
False
```

    - When the superclass is abstract, the subclass must implement all abstract methods from the superclass, without changing the public interface of those methods
- Inheritance serves as a form of *contract*:
    - The implementer of the subclass must implement the methods from the abstract superclass
    - Any user of the subclass may assume that they can call the superclass methods on instances of the subclass
- Because Stack1 and Stack2 are both subclasses of Stack, we expect them to implement all the stack methods

- They might also implement additional methods that are unique to each subclass (i.e. *not* shared)

Writing Polymorphic Code Using Inheritance

- Consider the following code that operates on a stack:

```
def push_and_pop(s: Stack, item: Any) -> None:
    """Push and pop the given item onto the stack s."""
    s.push(item)
    s.pop()
```

  - We use the abstract data class Stack as the type annotation, to indicate that our function push_and_pop can be called with *any* instance of any Stack subclass
  - We can pass a Stack1 or Stack2 object to the push_and_pop function because they both inherit from Stack

```
>>> s1 = Stack1()
>>> push_and_pop(s1)  # This works
>>> s2 = Stack2()
>>> push_and_pop(s2)  # This also works
```

- There are 3 versions of push: Stack.push, Stack1.push, Stack2.push
  - When the Python interpreter evaluates s.push(item), it first computes type(s).
    - i.e. type(s1) is Stack1, type(s2) is Stack2
  - The Python interpreter then looks in that class for a push method and calls it, passing in s for the self argument
- We say that the Python interpreter *dynamically looks up* (or *resolves*) the s.push/.pop method, because the actualmethodf called by s.push/.pop changes depending on the argument passed to push_and_pop
- We say that the push_and_pop function is *polymorphic*
  - *Polymorphic* – being able to take as inputs values of different concrete data type and select a specific method based on the type of input
- Inside the push_and_pop function, *neither* one of below would work
  - Stack.push(s, item)
    - Raises a NotImplementedError
  - Stack1.push(s, item)
    - Only works on the Stack1 instances, but not any other Stack subclasses
  - Both make push_and_pop no longer polymorphic

Application: Running Timing Experiments on Stack Implementation

- We can use polymorphism to help us measure the performance of each implementation (i.e. Stack1 and Stack2)

- We are calling the same function for both timing experiments (of Stack1 and of Stack2)

9.9 The Object Superclass

Intro

- object is an *ancestor class* of every other class
  - *Ancestor class* – parent class, or parent of a parent class
- Whenever we define a new class (including data classes), if we do not specify a superclass in parentheses, object is the *implicit* superclass

The Object Special Methods

- The object class defines several special methods as part of its shared public interface, including:
  - __init__(self, …)
    - The initializer
  - __str__(self)
    - Returns a str representation of the object

Method Inheritance

- The object class is *not abstract* and implements each of the special methods
- Here, where the superclass is a concrete class, inheritance is used not just to define a shared public interface, but also to provide *default implementations* for each method in the interface
- Suppose we create a dummy class with a completely empty body:

```
class Donut:
    """A donut."""
```

- This class inherits the object.__init__ method, which allows us to create new Donut instances

```
>>> donut = Donut()
>>> type(donut)
<class '__main__.Donut'>
```

- Similarly, this class inherits the object.__str__ method, which returns a string that states the class name and memory location of the object

```
>>> d = Donut()
>>> d.__str__()
'<__main__.donut object at 0x7fc299d7b588>'
```

- We can use the built-in dir function to see all of the special methods that Donut has inherited form object

```
>>> dir(Donut)
['__class__', '__delattr__', '__dict__', (the rest is omitted by me)]
```

- The special methods are often called by other functions or parts of Python syntax
  - We have already seen how the __init__ method is called when a new object is initialized
  - The __str__ method is called when we attempt to convert an object to a string by calling str on it

```
>>> d = Donut()
>>> d.__str__()
'<__main__.Donut object at 0x7fc299d7b588>'
>>> str(d)
'<__main__.Donut object at 0x7fc299d7b588>'
```

  - The built-in print function first converts its arguments into strings using their __str__ methods, and then prints out the resulting text

Method Overriding

- Every time we've defined our own __init__ in a class, we have *overridden* the object.__init__ method
- We say that a class C *overrides* a method m when the method m is defined in the superclass of C, and is also given a concrete implementation in the body of C
- When we defined a custom exception class

```
class EmptyStackError(Exception):
    """..."""
    def __str__(self) -> str:
        """..."""
        return 'pop may not be called on an empty stack'
```

  - This class *overrode* the __str__ method to use its own string representation, which is displayed when this exception is raised.

10.1 The Problem Domain: Food Delivery Networks

What Is a Problem Domain?

- *Problem domain* – collection of knowledge about a specific field, phenomenon, or discipline, and an understanding of the goals, problems, deficiencies, and/or desired improvements within that area
  - We've touched on a wide array of problem domains, such as:

        - Tracking marriage records in Toronto
        - Modelling the spread of infectious diseases
        - Generating course timetables as U of T students
        - Cryptography
- As we dove into cryptography, we learned about:
    - Terminology and definitions
        - Symmetric-key and public-key cryptosystems
        - Encryption and decryption
        - Various existing cryptosystems
    - Concepts and skills
        - Proving that a cryptosystem is correct
        - Justifying the security of a cryptosystem based on the presumed hardness of mathematical problems like Integer Factorization
    - The context and history
        - Ancient cryptosystems
        - How cryptography is applied to Internet communications
- It was the domain-specific knowledge we learned that explained *how* we came up with the cryptography algorithms and why they are correct

<u>Introducing Hercules</u>

- We want to launch a Hercules app that allows people to order groceries and meals from grocery stores and restaurants, and arrange for couriers to make deliveries right to their front doors
- When designing and implementing this app, we need to consider:
    - How restaurants will register with the app and post menus
    - How customers will register with the app to browse restaurants and place orders
    - How couriers will register with the app to claim orders and deliver them from restaurants to customers
    - … and more

<u>Food Delivery As a System</u>

- We can view food delivery in Toronto as a *system*
    - *System* – a group of entities (or agents) that interact with each other over time
- The first part of creating a computational model for such a system is to design and implement the various entities in the system
    - In the case of the Hercules Ltd., these are entities like couriers and customers placing orders
- The entities in a system change over time

    - i.e. new people sign up and place food orders
- The second part of our computational model is a *simulation* that uses randomness to generate events that cause the system to change over time
    - i.e. our food delivery simulation will specify how often customers place an order, taking into account that some times of day are busier than others
- A well-designed simulation allows the programmer to start with a simple model and extend and tweak it in response to new domain-specific knowledge

10.2 Object-Oriented Modelling of Our Problem Domain

<u>Entities in a Food Delivery System</u>

- We use two strategies for picking our relevant entities:
    - Identify different roles that people/group play in the domain
        - Each "role" is likely an entity
            - i.e. customer, courier, restaurant
    - Identify a bundle of data that makes sense as a logical unit
        - Each "bundle" is likely an entity
            - i.e. an order is a bundle of related information about a user's food request
- A standard approach is to create a class to represent each of these entities
    - We can start with a data class and turn it into a general class if we need a more complex design (i.e. to add methods, including the initializer, or mark attributes as private)

```
@dataclass
class Restaurant:
    """A place that servers food."""

@dataclass
class Customer:
    """A person who orders food."""

@dataclass
class Courier:
    """A person who delivers food orders from restaurants to customers."""

@dataclass
class Order:
    """A food order from a customer."""
```

Designing the Restaurant Data Class

- We need a way to identify each restaurant: its *name*
    - We'll use a str to represent it
- A user needs to see what food is available to order, so we need to store a *food menu* for each restaurant
    - Since it has a few different options, we'll use a dict that maps the names of dishes (strs) to their price (floats)
- Couriers need to know where restaurants are in order to pick up food orders, and so we need to store a *location* for each restaurant
    - We could store its address as a str
    - We could also store the latitude and longitude (a tuple of floats)
- Each of these three pieces of information (restaurant name, food menu, location) are appropriate *attributes* for the restaurant

```
@dataclass
class Restaurant:
    """A place that serves food.

    Instance Attributes:
        - name: the name of the restaurant
        - address: the address of the restaurant
        - menu: the menu of the restaurant with the name of the dish
        mapping to the price
        - location: the location of the restaurant as (latitude, longitude)

    Representation Invariants:
        - all(self.menu[item] >= 0 for item in self.menu)
        - -90 <= self.location[0] <= 90
        - -180 <= self.location[1] <= 180
    """
    name: str
    address: str
    menu: Dict[str, float]
    location: Tuple[float, float]
```

- Since the menu is a compound data type, we could have created a completely separate Menu data class
- Each new class we create introduces a little more complexity into our program, and for a relatively simple class for a menu, this additional complexity does not worth it

- We could have used a dictionary to represent a restaurant instead of a Restaurant data class
    - This would have reduced on area of complexity, but introduced another
        - i.e. the "valid" keys of a dictionary used to represent a restaurant

Designing the Order Data Class

- An order must track the *customer* who placed the order, the *restaurant* where the food is being ordered from, and the *food items* that are being ordered
- An order should have an associated courier who has been assigned to deliver the order
- We'll also keep track of when the order was created, and when the order is completed
- The associated courier and the time when the order is completed might only be assigned values after the order has been created
    - We use a default value None to assign to these two instance attributes when an Order is first created
    - We could implement this by converting the data class to a general class and writing our own __init__ method
    - We'll take advantage of a new feature with data classes: the ability to specify default values for an instance attribute after the type annotation

```
@dataclass
class Order
        """A food order from a customer.

        Attributes:
            - customer: the name of the customer who placed this order
            - restaurant: the name of the restaurant the order is place for
            - food_items: Dict[str, int]
            - start_time: datetime.datetime
            - courier: Optional[Courier] = None
            - end_time: Optional[datetime.datetime] = None
```

- The line courier: Optional[Courier] = None is how we define an instance attribute Courier with a default value of None
    - The type annotation Optional[Courier] means tha this attribute can either bge None or a Courier instance
    - Similarly, the end_time attribute must be either None (its initial value) or a datetime.datetime value
- Here is how we could use this class
    - Note: Customer is currently an empty data class, and so is instantiated simply as Customer()

```
>>> david = Customer()
>>> mcdonalds = Restaurant(name='McDonalds', address='160 Spadina Ave',
menu={'fries': 4.5}, location=(43.649, -79.397))
>>> order = Order(customer=david, restaurant=mcdonalds, food_items={'fries':
10}, start_time=datetime.datetime(2020, 11, 5, 11, 30))

>>> order.courier is None  # Illustrating default values
True
>>> order.end_time is None
True
```

Class Composition

- Classes can be "nested" within each other through their instance attributes
    - i.e. our Order data class has attributes which are instances of other classes we have defined (Customer, Restaurant, and Courier)
- The relationship between Order and these other classes is called *class composition*, and is fundamental to object-oriented design
- We use class composition to represent a "has a" relationship between two classes
    - i.e. "an Order has a Customer"

10.3 A "Manager" Class

Intro

- We can create a new manager class whose role is to keep track of all the entities in the system and to mediate the interactions between them (like a customer placing a new order)
- The FoodDeliverySystem will store (and have access to) every customer, courier, and restaurant represented in our system

```
class FoodDeliverySystem
    """A system that maintains all entities (restaurants, customers, couriers,
    and orders).

    Public Attributes:
        - name: the name of this food delivery system

    Representation Invariants:
        - self.name != ''
        - all(r == self._restaurants[r].name for r in self._restaurants)
```

```
        - all(c == self._customers[c].name for c in self._customers)
        - all(c == self._couriers[c].name for c in self._couriers)
    """
    name: str

    # Private Instance Attributes:
    #     - _restaurants: a mapping from restaurant name to Restaurant
    #     object. This represents all the restaurants in the system.
    #     - _customers: a mapping from customer name to Customer
    #     object. This represents all the customers in the system.
    #     - _couriers: a mapping from courier name to Courier object.
    #     This represents all the couriers in the system.
    #     - _orders: a list of all orders (both open and completed orders).

    _restaurants: Dict[str, Restaurant]
    _customers: Dict[str, Customer]
    _couriers: Dict[str, Courier]
    _orders: List[Order]

    def __init__(self, name: str) -> None:
        """Initialize a new food delivery system with the given company
        name.

        The system starts with no entities.
        """
        self.name = name
        self._restaurants = {}
        self._customers = {}
        self._couriers = {}
        self._orders = []
```

Changing State

- So far, we have modelled the *static* properties of our food delivery system, that is, the attributes that are necessary to capture a particular snapshot of the state of the sytem at a specific moment in time
- Adding entities
    - We can define simple methods to add entities to the system
      class FoodDeliverySystem:

…

def add_restaurant(self, restaurant: Restaurant) -> bool:

"""Add the given restaurant to this system.

Do NOT add the restaurant if one with the same name already exists.

Return whether the restaurant was successfully added to this system.

"""

if restaurant.name in self._restaurants:

return False

else:

self._restaurants[restaurant.name] = restaurant

return True

def add_customer(self, customer: Customer) -> bool"

"""Add the given customer to this system.

Do NOT add the customer if one with the same name already exists.

Return whether the courier was successfully added to this system.

Similar implementation to add_restaurant

def add_courier(self, courier: Courier) -> bool:

"""Add the given courier to this system.

Do NOT add the courier if one with the same name already exists.

Return whether the courier was successfully added to this system.

Similar implementation to add_restaurant

- Placing orders
  - When a customer places an order, a chain of events is triggered:
    - 1. The order is sent to the restaurant and to the assigned courier.
    - 2. The courier travels to the restaurant and picks up the food, and then brings it to the customer.
    - 3. Once the courier has reached their destination, they indicate that the delivery has been made.

class FoodDeliverySystem:

…

def place_order(self, order: Order) -> None:

"""Record the new given order.

Assign a courier to this new order (if a courier is available).

```
        Preconditions:
            - order not in self.orders
        """

    def complete_order(self, order: Order) -> None:
        """Mark the given order as complete.
        Make the courier who was assigned this order available to take a
        new order.

        Preconditions:
            - order in self.orders
        """
```

- FoodDeliverySystem.place_order would be responsible for both recording the order and assigning a courier to that order
- FoodDeliverySystem.complete_order marks the order as complete and un-assigning the courier so that they are free to take a new order

10.4 Food Delivery Events

The Event Interface

- We'll define abstract Event class with subclasses NewOrderEvent and CompleteOrderEvent to represent different kinds of events

```
class Event:
    """An abstract class representing an event in a food delivery
    simulation."""

    def handle_event(self, system: FoodDeliverySystem) -> None:
        """Mutate the given food delivery system to process this
        event."""
        raise NotImplementedError
```

- The abstract method handle_event is how we connect each event to a change in a food delivery system
- Each Event subclass is responsible for implementing handle_event based on the type of change the subclass represents
  - The NewOrderEvent.handle_event method should add a new order to the system

Common Instance Attributes

- A superclass can declare *public instance attributes* that its subclasses must have in common
- For our Event class, we can establish that all event subclasses will have a timestamp indicating when the event took place

```
class Event:
    """…

    Instance Attributes:
        - timestamp: the start time of the event
    """
    timestamp: datetime.datetime

    def __init__(self, timestamp: datetime.datetime) -> None:
        """Initialize this event with the given timestamp."""
        self.timestamp = timestamp


class NewOrderEvent(Event):
    """An event where a customer places an order for a restaurant."""
```

  - Since subclasses inherit all the methods from their superclass, we *must* provide a datetime.datetime object as the first argument when creating a new NewOrderEvent object

```
>>> e = NewOrderEvent(datetime.datetime(2020, 9, 8))
>>> e.timestamp
datetime.datetime(2020, 9, 8, 0, 0)
```

Subclass-Specific Attributes

- We often make the subclass-specific attributes private, to avoid changing the public interface declared by the abstract superclass
- We do *not* need to repeat the documentation for the timestamp attribute

```
class NewOrderEvent(Event):
    """An event representing when a customer places an order at a
    restaurant."""
    # Private Instance Attributes:
    #     - _order: the new order to be added to the FoodDeliverySystem
    _order: Order

    def __init__(self, order: Order) -> None:
```

```
        Event.__init__(self, order.start_time)
        self._order = order
```

- Now, whenever we call NewOrderEvent.__init__, Python also calls Event.__init__
  - This classes all shared instance attributes from Event to be "inherited" by the NewOrderEvent subclass

- When inheriting from a class that defines its own initializer:
  - 1. The initializer of a subclass must call the initializer of its super calss to initialize all common attributes
  - 2. The initializer of a subclass is responsible for initializing any additional attributes that are specific to that subclass

Implementing NewOrderEvent.handle_event

```
class NewOrderEvent(Event):
    """..."""
    ...
    def handle_event(self, system: FoodDeliverySystem) -> None:
        "Mutate system by placing an order."""
        system.place_order(self._order)
```

Implementing Other Event Subclass

- Similar to NewOrderEvent, but the initializer takes an explicit datetime.datetime argument to represent when the given order is completed

```
class CompleteOrderEvent(Event):
    """When an order is delivered to a customer by a courier."""
    # Private Instance Attributes:
    #       - _order: the order to be completed by this event
    _order: Order

    def __init__(self, timestamp: datetime.datetime, order: Order) -> None:
        Event.__init__(self, timestamp)
        self._order = order

    def handle_event(self, system: FoodDeliverySystem) -> None:
        """Mutate the system by recording that the order has been
        delivered to the customer."""
        system.complete_order(self._order, self.timestamp)
```

Event Generation

- Processing one event can cause other events to occur
    - i.e. when we process a NewOrderEvent, we expect that at some point in the future, a corresponding CompleteOrderEvent will occur
- We can change the return type of handle_event from None to List[Event], where the return value is a list of the events *caused* by the current event
    - NewOrderEvent would return a list containing a CompleteOrderEvent
        - If there are no available couriers, we would *not* want it to return a CompleteOrderEvent

```
class NewOrderEvent(Event):
    …
    def handle_event(self, system: FoodDeliverySystem) -> List[Event]:
        """…"""
        success = system.place_order(self._order)
        if success:
            completion_time = self.timestamp +
            datetime.timedelta(minutes=10)
            return [CompleteOrderEvent(completion_time,
            self._order)]
        else:
            self._order.start_time = self.timestamp +
            datetime.timedelta(minutes=5)
            return [NewOrderEvent(self._order)]
```

Returning No Events

- Our CompleteOrderEvent does not cause any new events to happen
    - Returns an empty list

A New Event Type

- This new event type will represent a random generation of new orders over a given time period, which we'll use to drive our simulation

```
class GenerateOrdersEvent(Event):
    """An event that causes a random generation of new orders.

    Private Representation Invariants:
        - self.._duration > 0
    """
    # Private Instance Attributes:
```

```
#         - _duration: the number of hours to generate orders for
_duration: int

def __init__(self, timestamp: datetime.datetime, duration: int) -> None:
    """Initialize this event with timestamp and the duration in hours.

    Preconditions:
        - duration > 0
    """

def handle_event(self, system: FoodDeliverySystem) -> List[Event]:
    """Generate new orders for this event's timestamp and
    duration."""
    events = []
    while…:
        # Create a randomly-generated NewOrderEvent
        new_order_event = …
        events.append(new_order_event)

    return events
```

10.5 Creating a Discrete-Event Simulation

The Main Simulation Loop

- *Discrete-event simulation* – simulation driven by individual events occurring at specified periods of time
- A discrete-event simulation runs as follows:
    - 1. It keeps track of a collection of events, which begins with some initial events. The collection is a *priority queue*, where an event's priority is its timestamp (earlier timestamps mean higher priority)
    - 2. The highest-priority event (i.e. the one with the earliest timestamp) is removed and processed. Any new events it generates are added to the priority queue
    - Step 2 repeats until there are no events left
- Assuming we have a *priority queue* implementation called EventQueueList, here is how we could write a simple function that runs this simulation loop:

```
def run_simulation(initial_events: List[Event], system: FoodDeliverySystem) ->
None:
```

```
    # Initialize an empty priority queue of events
    events = EventQueueList()
    for event in initial_events:
        events.enqueue(event)

    # Repeatedly remove and process the next event
    while not events.is_empty():
        event = events.dequeue()

        new_events = event.handle_event(system)
        for new_event in new_events:
            events.enqueue(new_event)
```

- Our run_simulation function is *polymorphic*
  - It works regardless of what Event instances it's given in its initial_events parameter, or what new events are generated and stored in new_events
  - Our function needs to be able to call the handle_event method on each event object

A Simulation Class

```
class FoodDeliverySimulation
    """A simulation of the food delivery system."""
    # Private Instance Attributes:
    #     - _system: The FoodDeliverySystem instance that this simulation uses
    #     - _events: A collection of the events to process during the simulation
    _system: FoodDeliverySystem
    _events: EventQueue

    def __init__(self, start_time: datetime.datetime, num_days: int,
    num_couriers: int, num_customers: int, num_restaurants: int) -> None:
        """Initialize a new simulation with the given simulation parameters.

        start_time: the starting time of the simulation
        num_days: the number of days that the simulation runs
        num_couriers: the number of couriers in the system
        num_customers: the number of customers in the system
        num_restaurants: the number of restaurants in the system
        """
        self._events = EventQueueList()
```

```
        self._system = FoodDeliverySystem()

        self._populate_initial_events(start_time, num_days)
        self._generate_system(num_couriers, num_customers, num_restaurants)

    def _populate_initial_events(self, start_time: datetime.datetime, num_days: int)
    -> None:
        """Populate this simulation's Event priority queue with
        GenerateOrdersEvents
        One new GenerateOrderEvent is generated per day, starting with
        start_time and repeating num_days times.
        """

    def _generate_system(self, num_couriers: int, num_customers: int,
    num_restaurants: int) -> None:
        """Populate this simulation's FoodDeliverySystem with the specified
        number of entities.
        """

    def run(self) -> None:
        """run this simulation.
        """
        while not self._events.is_empty():
            event = self._events.dequeue()

            new_events = event.handle_event(self._system)
            for new_event in new_events:
                self._events.enqueue(new_event)
```

- Key items to note in this (incomplete) implementation:
  - The run_simulation method has been renamed to simply run, since it's a method in the FoodDeliverySimulation class
  - The local variable events and parameter system from the function are now instance attributes for the FoodDeliverySimulation class, and have been moved out of the run method entirely. It's the job of the FoodDeliverySimulation.__init__ to initialize these objects
  - The initializer takes in several parameters representing *configuration values* for the simulation. It then uses these values in two helper methods to initialize the _system and _events objects. These methods are marked private (named with a

leading underscore) because they're only meant to be called by the initializer, and not code outside of the class

- To use the FoodDeliverySimulation class:

```
>>> simulation = foodDeliverySimulation(datetime.datetime(2020, 11, 30), 7, 4, 100, 50)
>>> simulation.run()
```

Populating Initial Events

- The key idea for our first helper method is that given a start time and a number of days, our initial events will be a series of GenerateOrderEvents that will generate NewOrderEvents when they are processed

```
def _populate_initial_events(self, start_time: datetime.datetime, num_days: int) -> None:
    """..."""
    for day in range(0, num_days):
        # 1. Create a GenerateOrderEvent for the given day after the start
        # time
        # 2. Enqueue the new event
```

Populating the System Entities

- Currently, only *orders* are dynamic in our system; the restaurants, customers, and couriers do not change over time
- The easiest way to populate these 3 entity times is to randomly generate new instances of each of these classes:

```
def _generate_system(self, num_couriers: int, num_customers: int, num_restaurants: int) -> None:
    """..."""
    for i in rtange(0, num_customers):
        location = _generate_location()
        customer = Customer(f'Customer {i}', location)
        self._system.add_customer(customer)

    # Couriers and Restaurants are similar
    ...
# Outside the class: helper for generating random locations in Toronto
TORONTO_COORDS = (43.747743, 43.691170, -79.644951, -79.176646)

def _generate_location() -> Tuple[float, float]:
```

“””Return a randomly-generated location (latitude, longitude) within the Toronto bounds.
“””
return (random.uniform(TORONTO_COORDS[0], TORONTO_COORDS[1]), random.uniform(TORONTO_COORDS[2], TORONTO_COORDS[3]))