OOP Design and Specification

ThanhVu (Vu) Nguyen

September 27, 2024 (latest version available on nguyenthanhvuh.github.io/class-oo/oop.pdf)

Preface

Contents

Chapter 1

Introduction

This book will guide you through the fundamentals of constructing high-quality software using a modern **object-oriented programming** (OOP) approach. We will use *Python* for demonstration, but the concepts can be applied to any object-oriented programming language. The goal is to develop programs that are reliable, efficient, and easy to understand, modify, and maintain.

1.1 Decomposition

As the size of a program increases, it becomes essential to *decompose* the program into smaller, independent programs (or functions or modules). This decomposition process allows for easier management of the program, especially when multiple developers are involved. This makes the program easier to understand and maintain.

Decomposition is the process of breaking a complex program into smaller, independent, more manageable programs, i.e., "divide and conquer". It allows programmer to focus on one part of the problem at a time, without worrying about the rest of the program.

Example Fig. 1.1 shows a Python implementation of *Merge Sort*, a classic example of problem decomposition. It breaks the problem of sorting a list into simpler problems of sorting smaller lists and merging them.

1.2 Abstraction

Abstraction is a key concept in OOP that allows programmers to hide the implementation details of a program and focus on the essential features. By decoupling the **what** (the behavior specification) from the **how** (the actual implementation), programmers could focus on higher-level design and reuse code more effectively. In

```python
def merge_sort(lst):
    if len(lst) <= 1:
        return lst

    mid = len(lst) // 2
    left = merge_sort(lst[:mid])
    right = merge_sort(lst[mid:])
    return merge(left, right)
```

```python
def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Fig. 1.1: Decomposition example: Mergesort

```python
class Mammal:
    def __init__(self, name):
        self.name = name

    def speak(self): pass

class Dog(Mammal):
    def speak(self):
        """
        EFFECTS: Return the sound of a dog.
        """
        return "Woof!"

class Cat(Mammal):
    def speak(self):
        """
        EFFECTS: Return the sound of a dog.
        """
        return "Meow!"
```

Fig. 1.2: Decomposition example: Mergesort

an OOP language such as Python, you can abstract problems by creating functions, classes, and modules that hide the underlying implementation details.

Example Fig. 1.2 demonstrates an abstraction for different types of mammals. Mammals such as Dog and Cat share common behaviors such as making noise (speak). We can create a class `Mammal` that defines these common behaviors, and then subclasses `Dog` and `Cat` that inherit from `Mammal` and define their own unique behaviors. These are abstract data types that allow us to work with mammals. Also notice the specification (e.g., `REQUIRES`) in the comments that describe what the method does, not how it does it.

Chapter 2

Procedural Abstraction

Procedural abstraction is a fundamental concept in programming that allows developers to create functions (methods) that hide the implementation details of a program. By abstracting away the details, developers can focus on the essential features of the program, making it easier to understand, modify, and maintain.

By separating procedure definition and invocation, we make two important methods of abstraction: abstraction by parameterization and abstraction by specification.

Abstraction by Parameterization This generalizes a function by using *parameters*. This allows the function to be used with different input values, making it more versatile and reusable. Fig. 2.1 shows an example of abstract parameterization. The `cal_area` function calculates the area of a rectangle given its length and width, which are passed as parameters.

```
def cal_area(length, width):
    return length * width

# can be used with different values for length and width.
area1 = cal_area(5, 10)
area2 = cal_area(7, 3)
```

Fig. 2.1: Example: Abstract Parameterization

Abstraction by Specification This specifies on what the function does (e.g., sorting), instead of how it does it (e.g., using quicksort or mergsort algorithms, implemented in C). By defining a function's behavior through *specifications*, developers can implement the function in different ways as long as it fulfills the specifications. Similarly, the user can use the function without knowing the implementation details.

Fig. 2.2 shows an example of abstraction by specification. The `exists` method return true if the `target` item is found in a list of sorted `items`. The user only needs to provide a sorted list and a target, but does not need to know what algorithm is used or implemented to determine if the item exists in the list.

```
def exists(items:List[int], target:int) -> bool:
    """
    Find an item in a list of sorted items.

    Pre: List of sorted items
    Post: True if the target is found, False otherwise.
    """
    ...

# The user only needs to know that this function checks
# for the existence of an item in a sorted list.
# They don't need to know the search algorithm or implementation.
```

Fig. 2.2: Abstraction by Specification

2.1 Specifications

We define abstractions through specifications, which describe what the abstraction is intended to do rather than how it should be implemented. This allows specifications to be much more concise and easier to read than the corresponding code.

Specifications which can be written in either *formal* or *informal languages*. Formal specifications have the advantage of being precise and unambiguous. However, in practice, we often use informal specifications, describing the behavior of the abstraction in plain English (e.g., the `sorting` example in Fig. 2.2). Note that a specification is not a programming language or a program. Thus, our specifications won't be written in code (e.g., in Python or Java)

2.1.1 Specifications of a Function

The specification of a function consists of a *header* and a *description* of its behavior. The header gives the signature of the function, including its name, parameters, and return type. The description describes the function's behavior, including its preconditions and postconditions.

Header The header provides the *name* of the function, the number, order, and types of its *parameters* (inputs), and the type of its return value (output). For instance, the headers for the `sort_items` function in Fig. 2.2 and the `cal_area` function in Fig. 2.1 are as follows

```
def exists(items: list) -> bool: ...
def calc_area(length: float, width: float) -> float: ...
```

Note that in a language like Java, the header also provides *exceptions* that the function may throw.

Preconditions and Postconditions A typical function specification in an OOP language such as Python includes: *Preconditions* (also called the "requires" clause)

and *Postconditions* (also called the "effects" clause). Preconditions describe the conditions that must be true before the function is called. Typically these state the constraints or assumptions about the input parameters. If there are no preconditions, the clause is often written as `None`.

Postconditions, under the assumption that the preconditions are satisfied, describe the conditions that will be true after the function is called. These typically state the expected results or outcomes of the function. Moreover, they often describe the relationship between the inputs and outputs.

The clauses are usually written as *comments* above the function definition, making them easily accessible within the code.

```
def calc_area(length: float, width: float) -> float:
    """
    Calculates the area of a rectangle given its length and width.

    Pre: None
    Post: The area of the rectangle.
    """
    ...
```

For example, the specification of the `calc_area` function in Fig. 2.1 has (i) no preconditions and (ii) the postcondition that the function returns the area of a rectangle given its length and width. Similarly, the `exists` function in Fig. 2.2 has the specification that given a list of sorted items (precondition), it returns true if the item is found in the list, and false otherwise (postcondition). Note how the specification is written in plain English, making it easy to understand for both developers and users of the function.

Modifies Another common clause in a function specification is *modifies*, which describes the inputs that the function modifies. This is particularly useful for functions that modify their input parameters.

```
def add_to_list(input_list, value):
    """
    Adds a value to the input list.

    Pre: None
    Post: Value is added to the input list.
    Modifies: the input list
    """
    ...
```

2.1.2 In-class Exercise: User Equality

This exercise touches on some thorny issues with inheritance. There is a lot going on in this example, but it is a good exercise to understand the subtleties of inheritance.

1. First, look at the Javadoc to understand the behaviors `equals()` (while the specification is for Java, the idea is the same in Python).

```python
class User:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        if not isinstance(other, User):
            return False
        return self.name == other.name
```

Fig. 2.3: User class

```python
class SpecialUser(User):
    """Don't do this until you've done with User"""

    def __init__(self, name, id):
        super().__init__(name)
        self.id = id

    def __eq__(self, other):
        if not isinstance(other, SpecialUser):
            return False
        return super().__eq__(other) and self.id == other.id
```

Fig. 2.4: SpecialUser class

- Specifically, read carefully the *symmetric*, *reflexive*, and *transitive* properties of `equals()`.
- Ignore *consistency*, which requires that if two objects are equal, they remain equal.

2. For the `User` class in Fig. 2.3, does `equals()` satisfy the three equivalence relation properties? If not, what is the problem?
   - Come up with several concrete test cases (e.g., create various `User` instances) to check the properties.
   - If there is a problem, show the test case that demonstrates the problem.
   - Explain why the problem occurs and come up with a fix.
3. So the same analysis for the `SpecialUser` class in Fig. 2.4.

2.2 Designing Specifications

When designing specifications, it is important to consider several factors to ensure that the function is well-defined and can be used effectively. These factors include the *strength* of the pre- and post-conditions, whether the function is *total* or *partial*, and the *avoiding implementation details* in the specification.

2.2.1 Weak Pre-conditions

m For pre-conditions, we want as weak a constraint as possible to make the function more versatile, allowing it to handle a larger class of inputs. Logically, a condition x is weaker than another if it is *implied* by the other y, i.e., $y \implies x$, or that x's constraints are a superset of y's. For example, the condition $x \leq 5$ is weaker than $x \leq 10$ and the input list is not sorted is weaker than the list is sorted (which is weaker than the list that is both sorted and has no duplicates). The *weakest* precondition is *True*, which indicates no constraints on the input.

2.2.2 Strong Post-conditions

In contrast, for post-conditions, we want as strong a condition as possible to ensure that the function behaves as expected. A condition y is stronger than another condition x if y implies x, i.e., $y \implies x$, or that y's constraints are a strict subset of x's. For example, the condition $x \leq 10$ is stronger than $x \leq 5$ or that the input list is sorted is stronger than the list is not sorted.

2.2.3 Total vs Partial Functions

A function is *total* if it is defined for all legal inputs; otherwise, it is *partial*. Thus a function with no precondition is total, while a function with the strongest possible precondition is partial. Total functions are preferred because they can be used in more situations, especially when the function is used publicly or in a library where the user may not know the input constraints. Partial functions can be used when the function is used internally, e.g., a helper or auxiliary function and the caller is knowledgeable and can ensure its preconditions are satisfied.

The functions `calc_area` function in Fig. 2.1 and `add_to_list` in Fig. 2.2 are total because they can be called with any input. The `exists` function in Fig. 2.2 is partial because it only works with sorted lists.

Turning Partial Functions into Total Functions It is often possible to turn a partial function into a total function in two steps. First, we move preconditions into postconditions and specify the expected behavior when the precondition is not satisfied, e.g., throws an `Exception`. Second, we modify the function to satisfy the new specification, i.e., handling the cases when the preconditions are not satisfied. For example, the `exists` function in Fig. 2.2 is turned into the total function shown in Fig. 2.5.

2.2.4 In-class Exercise: Partial and Total Specifications for `tail`

Consider the following code:

```
def tail(my_list):
    result = my_list.copy()
```

```
def exists(items: List[int], target: int) -> bool:
    """
    Find an item in a list of sorted items.

    Pre: True
    Post: If the input items are not sorted, raise an exception.
          Return True if the item is found, False otherwise.


    """

    if not is_sorted(items):
        raise Exception(...)
```

Fig. 2.5: Total Specification for the program in Fig. 2.2

```
    result.pop(0)
    return result
```

- What does the implementation of `tail` do in each of the following cases? You might want to see the Python document for `pop`. How do you know: Running the code or reading Python document?
  - `list = None`
  - `list = []`
  - `list = [1]`
  - `list = [1, 2, 3]`
- Write a *partial specification* for `tail`
- Rewrite the specification to be *total*. Use *exceptions* as needed.

2.2.5 No implementation details

The specification should not include any implementation details, such as the algorithm used or the data structures employed. This improves flexibility as it allows the function to be implemented in different ways as long as it satisfies the specification. For example, the `exists` function in Fig. 2.2 does not specify the search algorithm used to find the item in the list.

Some common examples to avoid include: the mentioning of specific data structures (e.g., arrays, indices), algorithms (e.g., quicksort or mergesort), and exceptions (e.g., related to IndexError). Also avoid specifications mentioning indices because this implies the use of arrays.

2.3 Exercise

2.3.1 Specification for Sorting

Write the specification for the generic `ascending_sort` method below. The specification should include preconditions and postconditions.

```
def ascending_sort(my_list):
    # REQUIRES/PRE:
    # EFFECTS/POST:
    ...
```

2.3.2 Specification of Binary Search

Come up with the specification for a *binary search* implementation whose header is given below. Remember for precondition you want something as *weak* as possible and for postcondition as *strong* as possible. Note that binary search returns the *location* (an non-neg integer) of the `target` value if found, and returns -1 if `target` is not found.

```
def binary_search(arr: List[int], target: int) -> int:
    """
    PRE/REQUIRES:
    POST/EFFECTS:
    """
    ...
```

2.3.3 Loan Calculator

Consider a function that calculates the number of months needed to pay off a loan of a given size at a fixed *annual* interest rate and a fixed *monthly* payment. For example, a $100,000 loan at an 8% annual rate would take 166 months to discharge at a monthly payment of $1,000, and 141 months to discharge at a monthly payment of $1,100. (In both cases, the final payment is smaller than the others; we round 165.34 up to 166 and 140.20 up to 141.) Continuing the example, the loan would never be paid off at a monthly payment of $100, since the principal would grow rather than shrink.

- Define a function satisfying the following specification:

```
def months(principal: int, rate: float, payment: int) -> int:
    """
    Calculate the number of months required to pay off a loan.

    param principal: Amount of the initial principal (in dollars)
    param rate: Annual interest rate (e.g., 0.08 for 8%)
    param payment: Amount of the monthly payment (in dollars)

    Requires/Pre: principal, rate, and payment all positive and
    payment is sufficiently large to drive the principal to zero.
    Effects/Post: return the number of months required to pay off the principal
    """
```

- The precondition is quite strong, which makes implementing the method easy. The key step in your calculation is to change the principal on each iteration with the following formula (which amounts to monthly compounding):

  ```
  new_principal = old_principal * (1 + monthly_interest_rate) - payment
  ```

- To make sure you understand the point about preconditions, your code is required to be *minimal*. Specifically, if it is possible to delete parts of your implementation and still have it satisfy the requirements, you'll earn less than full credit.

- *Total* specification: Now change the specification to *total* in which the post-condition handles violations of the preconditions using *exceptions*. In addition, provide a new implementation `month` that satisfies the new specification.

2.3.4 Partial and Total Functions

1. Write the *partial* specifications for the below two functions.
2. Modify the specifications to make the functions *total*.
3. Modify the implementations of the two functions to satisfy the total specification.

Recall that specifications do not deal with types (which are taken care by the function signature and enforced by the type system of compiler/interpreter). In other words, you do not need to worry about types here and can assume conditions about types are satisfied.

```
def divide(a:float, b:float) -> float:
    """
    PRE:
    POST:
    """
    return a / b

def get_average(numbers: list[float]) -> float:
    """
    PRE:
    POST:
    """
    total = sum(numbers)
    return divide(total, len(numbers))
```

Chapter 3

Data Abstraction

In 1974, Barbara Liskov and Stephen N. Zilles introduced the concept of Abstract Data Types (*ADTs*) in their influential paper "Programming with Abstract Data Types" as part of their work on the CLU programming language at MIT. ADTs revolutionized software design by separating the specification of a data type from its implementation. This meant that developers could define operations on a data structure (such as stacks or queues) without exposing the details of how the data was managed internally. This idea of data abstraction improved modularity, making programs easier to modify, extend, and maintain.

For her pioneering contributions to programming languages and system design, particularly through her work on ADTs and CLU, Barbara Liskov was awarded the Turing Award in 2008. Today, ADTs are a cornerstone of modern programming, underlying the concepts of encapsulation and modularity in modern OOP languages like Java, Python, and Rust.

3.1 Specifications of an ADT

The specification of ADT explains what the operations on the data type do, allowing users to interact with objects only via methods, rather than accessing the internal representation. As with functions (§2), the specification for an ADT defines its behaviors without being tied to a specific implementation.

Structure of an ADT In a modern OOP language such as Python or Java, data abstractions are defined using *classes*. Each class defines a name for the data type, along with its constructors and methods.

Fig. 3.1 shows an ADT class template in Python. It consists of three main parts. The *Overview* describes the abstract data type in terms of well-understood concepts, like mathematical models or real-world entities. For example, a stack could be described using mathematical sequences. The Overview can also indicate whether the objects of this type are *mutable* (their state can change) or *immutable*.

```
class DataType:
    """
    Overview: A brief description of the data type and its objects.
    """

    def __init__(self, ...):
        """
        Constructor to initialize a new object.
        """

    def method1(self, ...):
        """
        Method to perform an operation on the object.
        """
```

Fig. 3.1: Abstract Data Type template

The *Constructor* initializes a new object, setting up any initial state required for the instance. Finally, *methods* define operations users can perform on the objects. These methods allow users to interact with the object without needing to know its internal representation. In Python, `self` is used to refer to the object itself, similar to `this` in Java or C++.

Note that as with procedural specification (§2), the specifications of constructors and methods of an ADT do not include implementation details. They only describe what the operation does, not how it is done. Moreover, they are written in plain English as code comment.

3.1.1 Example: `IntSet` ADT

Fig. 3.2 gives the specification for an `IntSet` ADT, which represents unbounded set of integers. `IntSet` includes a constructor to initialize an empty set, and methods to insert, remove, check membership, get the size, and choose an element from the set. `IntSet` is also mutable, as it allows elements to be added or removed. *mutator* `insert` and `remmove` are mutator methods and have a `MODIFIES` clause. In contrast, `is_in`, `size`, and `choose` are *observer* methods that do not modify the object.

3.2 Implementing ADT

To implement an ADT, we first choose a *representation* (**rep**) for its objects, then design constructors to initialize it correctly, and methods to interact with and modify the rep. For example, we can use a `list` (or vector) as the rep of `IntSet` in Fig. 3.2. We could use other data structures, such as a `set` or `dict`, as the rep, but a list is a simple choice for demonstration.

To aid understanding and reasoning of the rep of an ADT, we use two key concepts: *representation invariant* and *abstraction function*.

```
class IntSet:
    """
    Overview: IntSets are unbounded, mutable sets of integers.
    This implementation uses a list to store the elements, ensuring no duplicates.

    """
    def __init__(self):
        """
        Constructor
        EFFECTS: Initializes this to be an empty set.
        """
        self.els = []  # the representation (list)

    def insert(self, x: int) -> None:
        """
        MODIFIES: self
        EFFECTS: Adds x to the elements of this set if not already present.
        """
        if not self.is_in(x): self.els.append(x)

    def remove(self, x: int) -> int:
        """
        MODIFIES: self
        EFFECTS: Removes x from this set if it exists. Also returns
        the index of x in the list.
        """
        i = self.find_idx(x)
        if i != -1:
            # Remove the element at index i
            self.els = self.els[:i] + self.els[i+1:]
        return i

    def is_in(self, x: int) -> (bool, int):
        """
        EFFECTS: If x is in this set, return True. Otherwise False.
        """
        return True if find_index(x) != -1 else False

    def find_idx(self, x:int)->int:
        """
        EFFECTS: If x is in this set, return its index. Otherwise returns -1.
        """
        for i, element in enumerate(self.els):
            if x == element:
                return i
        return -1

    def size(self) -> int:
        """
        EFFECTS: Returns the number of elements in this set (its cardinality).
        """
        return len(self.els)

    def choose(self) -> int:
        """
        EFFECTS: If this set is empty, raises an Exception.
        Otherwise, returns an arbitrary element of this set.
        """
        if len(self.els) == 0:
            raise Exception(...)
        return self.els[-1]  # Returns the last element arbitrarily

    def __str__(self) -> str:
        """
        Abstract function (AF) that returns a string representation of this set.
        EFFECTS: Returns a string representation of this set.
        """
        return str(self.els)
```

Fig. 3.2: The IntSet ADT

3.2.1 Representation Invariant (Rep-Inv)

Because the rep might not be necessarily related to the ADT itself (e.g., the list has different properties compared to a set), we need to ensure that our use of the rep is consistent with the ADT's behavior. To do this, we use *representation invariant* (**rep-inv**) to specify the constraints for the rep of the ADT to capture its behavior.

For example, the rep-inv for a stack is that the last element added is the first to be removed and the rep-inv for a binary search tree is that the left child is less than the parent, and the right child is greater. The rep-inv for our `IntSet` ADT in Fig. 3.2 is that all elements in the list are unique.

```
# Rep-inv:
# els is not null, only contains integers and has no duplicates.
```

The rep-inv must be preserved by all methods (more precisely, *mutator* methods). It must hold true before and after the method is called. The rep-inv might be violated temporarily during the method execution, but it must be restored before the method returns. For `IntSet` Notice that the mutator `insert` method ensures that the element is not already in the list before adding it.

The rep-inv is decided by the designer and specified in the ADT documentation as part of the specification (just like pre/post conditions) so that it is ensured at the end of each method (like the postcondition). Moreover, because rep-inv is so important, it is not only documented in comments but also checked at runtime. This is done by invoking a `repOK`, discussed later, method at the start and end of each method.

3.2.2 In-Class Exercise: Checking Rep-Invs

```
class Members:
    """
    Overview: Members is a mutable record of organization membership.
    AF: Collect the list as a set.

    Rep-Inv:
        - rep-inv1: members != None
        - rep-inv2: members != None and no duplicates in members.
        For simplicity, assume None can be a member.
    """

    def __init__(self):
        """Constructor: Initializes the membership list."""
        self.members = []  # The representation

    def join(self, person):
        """
        MODIFIES: self
        EFFECTS: Adds a person to the membership list.
        """
        self.members.append(person)

    def leave(self, person):
        """
```

```
        MODIFIES: self
        EFFECTS: Removes a person from the membership list.
        """
        self.members.remove(person)
```

1. Analyze these four questions for *rep-inv 1*.
   - Does `join()` maintain rep-inv?
   - Does `join()` satisfy its specification?
   - Does `leave()` maintain rep-inv?
   - Does `leave()` satisfy its specification?
2. Repeat for *rep-inv 2*.
3. Recode `join()` to make the verification go through. Which rep-invariant do you use?
4. Recode `leave()` to make the verification go through. Which rep-invariant do you use?

3.2.3 Abstraction Function (AF)

It can be difficult to understand the ADT by looking at the rep directly. For example, we might not be able to visualize or reason about a binary tree or a graph ADT when using list as the rep. To aid understanding, *abstraction function* (**AF**) provides a mapping between the rep and the ADT. Specifically, the AF maps from a *concrete state* (i.e., the `els` rep in Fig. 3.2) to an *abstract state* (i.e., the integer set). AF is also a *many-to-one* mapping, as multiple concrete states can map to the same abstract state, e.g., the list `[1, 2, 3]` and `[3, 2, 1]` both map to the same set `{1, 2, 3}`.

Just as with rep-inv, the AF is documented in the class specification. Modern OOP languages often provide methods implementing the AF, in particular developer overrides the `__str__` method in Python and `toString` in Java to return a string representation of the object. For example, the `__str__` method in Fig. 3.2 returns a string representation of the set.

3.2.4 In-Class Exercise: Stack ADT

In this exercise, you will implement a `Stack` ADT. A stack is a common data structure that follows the Last-In-First-Out (LIFO) principle. You will:

1. Choose a Representation (rep) for the stack.
2. Define a Representation invariant (rep-inv)
3. Write a `repOK` method

4. Provide the specifications of basic stack operations (`push`, `pop`, `is_empty`) and implement these methods accordingly.

5. Define an Abstraction Function (AF)

6. Implement `__str__()` to return a string representation of the stack based on the AF

3.3 Mutability vs. Immutability

An ADT can be either mutable or immutable, depending on whether their objects' values can change over time. An ADT should be immutable if the objects it models naturally have unchanging values, such as mathematical objects like integers, polynomials (Polys), or complex numbers. On the other hand, an ADT should be mutable if it models real-world entities that undergo changes, such as an automobile in a simulation, which might be running or stopped, or contain passengers, or if the ADT models data storage, like arrays or sets.

Immutability is beneficial because it offers greater safety and allows sharing of subparts without the risk of unexpected changes. Moreover, immutability can simplify the design by ensuring the object's state is fixed once created. However, immutable objects can be less efficient, as creating a new object for each change can be costly in terms of memory and time.

Converting from mutable to immutable Given a mutable ADT, it is possible to convert it to an immutable one by ensuring that the rep is not modified by any method. This can be achieved by making the rep private and only allowing read-only access to it. In Python, this can be done by using the `@property` decorator to create read-only properties. For example, the `els` list in Fig. 3.2 can be made read-only by defining a property method `elements` that returns a copy of the list.

```
class IntSet:
    def __init__(self):
        self.__els = []   # Private rep
    @property
    def self.els(self):
        return self.__els
```

Moreover, we need to convert mutator methods into observer methods, which make a copy of the rep, modify it, and return the modified rep object.

```
def insert_immutable(self, x: int) -> IntSet:
    new_set = self.els.copy()
    if not self.is_in(x):
        new_set = new_set.append(x)
    return new_set
```

If the mutator returns a value v, then our new method returns a tuple consisting of (i) the new rep object and the return the value v.

```
def remove_immutable(self, x: int): -> (IntSet, int):
    i = self.find_idx(x)
    new_set = self.els.copy()
    if i != -1:
        # Remove the element at index i
        new_set = self.els[:i] + self.els[i+1:]
    return (new_set, i)
```

If you do not want to return multiple values (e.g., like in Java), then you can create two methods, one for returning the value and the other for returning the new rep object. For example, a mutator `pop` method of a `Stack` would result into two methods: `pop2` returns the top element and `pop3` returns the new stack with the top element removed.

Finally, it is important that while it is possible to convert a mutable ADT to an immutable one as shown, mutability or immutability should be the property of the ADT type itself, not its implementation. Thus, it should be decided at the design stage and documented in the ADT specification.

3.3.1 In-class Exercise: Immutable Queue

Rewrite the mutable `Queue` implementation in Fig. 3.3 so that it becomes *immutable*. Keep the **rep** variables `elements` and `size`.

3.4 Exercise

3.4.1 Polynomial ADT

Use the Poly ADT in Fig. 3.4 to answer the following questions. Use the `Stack` ADT in Fig. B.1 as an example.

1. Part 1
   (a) Write an Overview that describes what `Poly` does. You must provide some examples to demonstrate (e.g., `Poly(2,3)` means what?).
   (b) Provide the specifications for all methods in the ADT.
   (c) Write the **rep** used in this code. Describe how this rep represents `Poly`.
   (d) Provide the **rep-inv** for the ADT. Note, this would be the constraints over the rep variable(s).
   (e) Write a `repOK` method that checks the rep-inv.
   (f) Describe the AF in this code. Use `__str__` to help.
2. Part 2
   (a) Introduce a fault (i.e. "bug") that breaks the **rep-inv**. Try to do this with a small (conceptual) change to the code. Show that the rep-invariant is broken with a concrete test case.

```
class Queue:
    """
    A generic Queue implementation using a list.
    """

    def __init__(self):
        """
        Constructor
        Initializes an empty queue.
        """
        self.elements = []
        self.size = 0

    def enqueue(self, e):
        """
        MODIFIES: self
        EFFECTS: Adds element e to the end of the queue.
        """
        self.elements.append(e)
        self.size += 1

    def dequeue(self):
        """
        MODIFIES: self
        EFFECTS: Removes and returns the element at the front of the queue.
        If the queue is empty, raises an IllegalStateException.
        """
        if self.size == 0:
            raise Exception(...)

        result = self.elements.pop(0)  # Removes and returns the first element
        self.size -= 1
        return result

    def is_empty(self):
        """
        EFFECTS: Returns True if the queue is empty, False otherwise.
        """
        return self.size == 0
```

Fig. 3.3: Mutable Queue

```
class Poly:
    def __init__(self, c=0, n=0):
        if n < 0:
            raise ValueError("Poly(int, int) constructor: n must be >= 0")
        self.trms = {}
        if c != 0:
            self.trms[n] = c

    def degree(self):
        if len(self.trms) > 0:
            return next(reversed(self.trms.keys()))
        return 0

    def coeff(self, d):
        if d < 0:
            raise ValueError("Poly.coeff: d must be >= 0")
        return self.trms.get(d, 0)

    def sub(self, q):
        if q is None:
            raise ValueError("Poly.sub: q is None")
        return self.add(q.minus())

    def minus(self):
        result = Poly()
        for n, c in self.trms.items():
            result.trms[n] = -c
        return result

    def add(self, q):
        if q is None:
            raise ValueError("Poly.add: q is None")

        non_zero = set(self.trms.keys()).union(q.trms.keys())
        result = Poly()
        for n in non_zero:
            new_coeff = self.coeff(n) + q.coeff(n)
            if new_coeff != 0:
                result.trms[n] = new_coeff
        return result

    def mul(self, q):
        if q is None:
            raise ValueError("Poly.mul: q is None")

        result = Poly()
        for n1, c1 in self.trms.items():
            for n2, c2 in q.trms.items():
                result = result.add(Poly(c1 * c2, n1 + n2))
        return result

    def __str__(self):
        r = "Poly:"
        if len(self.trms) == 0:
            r += " 0"
        for n, c in self.trms.items():
            if c < 0:
                r += f" - {-c}x^{n}"
            else:
                r += f" + {c}x^{n}"
        return r
```

Fig. 3.4: Polynomial ADT

(b) Analyzed your bug with respect to the method specifications of `Poly`. Are all/some/none of the specification violated?

(c) Do you think your fault is realistic? Why or why not?

3.4.2 Immutability

The below class `Immutable` is supposed to be immutable. However, it is not. Identify the issues and fix them. Note that in Python or Java, immutable types include `int`, `float`, `str`, `tuple`. and mutable types include `list` and `dict`.

1. Which of the lines (A–F) has a problem with immutability? Explain why by showing code example, i.e., show code involving problematic lines; show how that breaks immutability.

2. For each line that has a problem. Write code to fix it so that the class is immutable.

```
class Immutable:
    def __init__(self, mstr: str, mint: int, mlist: list[str]):
        self._mstr = mstr                       # Line A
        self._mint = mint                       # Line B
        self._mlist = mlist.copy()              # Line C


    def get_mstr(self) -> str: return self._mstr        # Line D
    def get_mint(self) -> int: return self._mint        # Line E
    def get_mlist(self) -> list[str]: return self._mlist # Line F
```

Chapter 4

Types

In 1999, NASA's Mars Climate Orbiter mission ended in failure due to a simple yet catastrophic software error. The spacecraft, which cost $125 million to build and launch, was launched on December 11, 1998 to study the Martian climate and atmosphere. After a 9-month journey, the spacecraft approached Mars on September 23, 1999, and was supposed to enter a stable orbit around Mars at an altitude of about 226 kilometers (140 miles) above the planet's surface. However, the spacecraft instead plunged much deeper into the Martian atmosphere, to an estimated altitude of 57 kilometers (35 miles), causing it to either burn up or crash on the surface, resulting in a complete loss of the mission.

The cause of the failure was a software error involving typing mismatch between imperial units (pounds-force) and metric units (newtons) in the software that controlled the spacecraft's thrusters. The software expected data in metric units, but the thruster data was provided in imperial units, leading to the incorrect trajectory calculations. This mismatch was not caught during testing, and the spacecraft was lost as a result. This failure not only cost NASA a significant financial investment but also set back the Mars exploration program.

4.1 Type Systems in OOP

In OOP, the type system forms the foundation for defining how ADT (§3) is represented and manipulated in a language. Type systems provide rules for assigning types to variables, expressions, functions, and objects, enabling the development of reliable and efficient software. A well-defined type system also enforces contracts between components, ensuring that data is used appropriately.

This chapter covers key concepts in the type system of OOP languages, particularly in the context of Python, where both static and dynamic typing coexist. We will explore topics like polymorphism, inheritance, dynamic dispatching, and more, discussing their motivation, core concepts.

```python
from abc import ABC, abstractmethod

class Mammal(ABC):
    """
    Abstract class
    """

    @abstractmethod
    def speak(self):
        raise NotImplementedError("Subclasses should implement this!")

class Dog(Mammal):
    def speak(self):
        return "Woof!"

    def bark(self):
        return "Bark!"

class Cat(Mammal):
    def speak(self): return "Meow!"

# Using polymorphism
def make_animal_speak(mammal: Mammal): return mammal.speak()

mammals = [Dog(), Cat()]
for m in mammals:
    print(make_animal_speak(m))
```

Fig. 4.1: Polymorphism

4.2 Polymorphism

Polymorphism is a cornerstone of OOP that allows objects of different types to be treated as objects of a common supertype. This facilitates flexibility in programming by enabling the use of a unified interface for different types of objects, reducing redundancy and increasing code reuse.

Fig. 4.1 shows an example of subtype polymorphism, where a `Mammal` class has two subclasses, `Dog` and `Cat`, each implementing the `speak` method differently. The `make_mammal_speak` function can then be used to make any mammal speak, regardless of its specific type.

4.3 Inheritance

Inheritance creates a hierarchical relationship between classes and allows a class to be a *subclass* or *subtype* of one other class (its *superclass* or *supertype*). Fig. 4.1 shows an example of inheritance. `Mammal` is the superclass of `Dog` and `Cat`. `Dog` and `Cat` are the subclasses of `Mammal` They override `speak` to provide a specific implementation. In addition to *overriding* the `speak` method in `Mammal`, `Dog` defines a new method `bark` that is specific to dogs.

This is an example of single inheritance, where a subclass can inherit from only one superclass. Python also supports multiple inheritance, where a subclass can inherit from multiple superclasses. For example, an `HybridVehicle` class could inherit from both `Car` and `BatteryVehicle` classes. However, multiple inheritance can lead to complex hierarchies and potential conflicts, so it should be used judiciously.

4.4 Abstract Class

OOP has two types of classes: *concrete* and *abstract* classes. Concrete classes provide a full implementation of the type while abstract classes provide at most a partial implementation of the type. Abstract classes cannot be instantiated (no objects) since some of their methods are not yet implemented (abstract methods). Abstract classes can have both abstract (to be implemented by subclasses) and concrete methods (already implemented or partially implemented).

In Python abstract classes are defined using the `abc` module, which provides the `ABC` class and the `abstractmethod` decorator. The `ABC` class is used as a base class for abstract classes, and the `abstractmethod` decorator is used to mark methods as abstract. In Fig. 4.1, `Mammal` is an abstract class, i.e., it cannot be instantiated, and contains an abstract method `speak` that its subclasses must implement. In Java, abstract classes are defined using the `abstract` keyword, e.g., `public abstract class Mammal`, and abstract methods are declared using the `abstract` keyword as well, e.g., `public abstract void speak();`.

4.5 Interface

Interface is a special type of abstract class that contains only abstract methods (no concrete methods). They define a specification that classes must adhere to, providing the methods that must be implemented by any class that implements the interface. Multiple classes can implement the same interface, allowing for polymorphism and flexibility in the design.

In Python, interfaces are not explicitly defined, but the concept can be implemented using abstract classes with only abstract methods. For example, the abstract class `Mammal` in Fig. 4.1 acts as an interface that specifies the `speak` method that all mammals must implement. In Java, interfaces are explicitly defined using the `interface` keyword, e.g., `interface Mammal`, and methods are declared without a body, e.g., `public void speak();`. A class can implement multiple interfaces, allowing for more flexibility in defining contracts between classes.

Comparable interface A good example of an interface is `Comparable`, which defines a single method `compare_to` that allows objects to be compared to each other. Any class that implements `Comparable` can be compared to other objects of the same type, enabling sorting and other operations that require comparison.

The code below demonstrates the use of the `Comparable` interface in Python. The `Number` class implements the `Comparable` interface by defining the `compare_to` method, which compares two `Number` objects based on their values. The `sort` function uses the `compare_to` method to sort a list of `Number` objects.

```
    from abc import ABC, abstractmethod
from typing import List

# Define a Comparable interface using ABC
class Comparable(ABC):
    @abstractmethod
    def compare_to(self, other: "Comparable") -> int:
        """Compares this object with another."""
        pass

# Implement Comparable in a concrete class
class Number(Comparable):
    def __init__(self, value: int):
        self.value = value

    def compare_to(self, other: "Number") -> int:
        if self.value < other.value:
            return -1
        elif self.value > other.value:
            return 1
        else:
            return 0

# Polymorphic sorting function that relies on the compare_to method
def sort(items: List[Comparable]) -> List[Comparable]:
    return sorted(items, key=lambda x: x.value)

# Usage
numbers = [Number(3), Number(1), Number(4), Number(2)]
sorted_numbers = sort(numbers)
print(sorted_numbers)  # Output: [1, 2, 3, 4]
```

4.5.1 Element Subtype vs Related Subtype

There are two types of subtypes: *element subtype* and *related subtype*. Element subtype relies on a common interface or abstract class, e.g., `Number` is an element subtype of `Comparable`. While this common approach allows for polymorphism, it requires all potential types must be pre-planned to fit the hierarchy.

On the other hand, a related subtype does not directly rely on a common interface or abstract class (which might be designed much later). Instead, this approach creates a related subtype that implement the desired interface and then adapts it to the existing hierarchy. The code below demonstrates the use of a related subtype, where `Price` is adapted to `PriceComparable`, which implements `Comparable`, to allow sorting of `Price` objects.

```
class Price:
    def __init__(self, amount: float):
        self.amount = amount
```

```
class PriceComparable(Comparable):
    def __init__(self, price: Price):
        self.price = price
    def compare_to(self, other: "PriceComparable") -> int:
        if self.price.amount < other.price.amount:
            return -1
        elif self.price.amount > other.price.amount:
            return 1
        else:
            return 0

# sorting using related subtype
prices = [Price(3.0), Price(1.0), Price(4.0), Price(2.0)]
price_comparators = [PriceComparable(p) for p in prices]
sorted_prices = sort(price_comparators)
```

4.6 Dynamic Dispatching

Dynamic dispatching refers to how a program selects which method to invoke when a method is called on an object. It allows the correct method to be invoked based on the *runtime type* of the object, even if the reference to the object is of a more general (superclass) type. This is particularly useful when working with inheritance and polymorphism, where subclasses override methods from a superclass. The distinction between dynamic dispatching and static dispatching lies in when the decision about which method to invoke is made—either at runtime (dynamic) or compile-time (static).

In Fig. 4.1 the `make_mammal_speak` method will invoke the `speak` method of the correct subclass based on the runtime type of the object. This is dynamic dispatching in action, where the method `speak` to be called is determined at runtime based on the actual type of the object. However, if we explicitly create a `Dog` instance and call `speak` on it, the method is statically dispatched, as the compiler knows the type of the object at compile-time and can directly call the correct method.

The code below demonstrates the difference between static and dynamic dispatching. The `Dog` object `d` is statically dispatched, while the `Mammal` object `m` is dynamically dispatched.

```
Dog d = Dog();
d.speak();  # Static dispatching

Mammal m = Dog();
m.speak();  # Dynamic dispatching
```

4.7 Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP) is a fundamental concept of object-oriented design, which ensures that objects of a subclass should be able to replace objects

of the superclass without altering the correctness of the program. LSP promotes proper design and enforces correct use of inheritance. Violating LSP can lead to unexpected behavior and errors in the program, as the assumptions made about the superclass may no longer hold for the subclass.

The main idea of LSP is that a subclass *is-a* superclass and can do everything the superclass can do, and can also do more. For example, a `Dog` is a `Mammal` and can speak like any mammal, but it can also bark, which is specific to dogs. This enables substitution of objects of the subclass for objects of the superclass, allowing for polymorphism and dynamic dispatching to work correctly. The `make_animal_speak` function in Fig. 4.1 demonstrates LSP by accepting any `Mammal` object and making it speak, regardless of its specific type.

Rules If `S` is a subtype of `T`, then objects of type `T` may be replaced with objects of type `S` without altering any of the desirable properties of the program. This means whenever you use `T`, you can use `S` instead. To achieve this, we must follow the following rules:

Signature Rule The signatures of methods of `S` must strengthen methods of `T`. In other words, the methods of `S` are a superset of the methods of `T`. Thus, if `T` has `n` methods, `S` also has `n` methods and additional ones (methods specific to `S`).

Method Rule The specification of `f'` strengthens that of `f`. This means that the preconditions of `f'` must be weaker or equal to the preconditions of `f`, i.e., `f'` accepts more inputs than `f`. The postconditions of `f` must be stronger or equal to that of `f`. This means that `f'` is more precise and specific than `f`.

Property Rule The subtype must preserve all properties of the supertype. For example, the rep-invariant of the subtpe `S` must be stronger or equal to that of the supertype `T`. This means `S` should maintain or strengthen the properties (including rep invariants) of `T`

4.7.1 In-Class Exercise: Bank Account

4.8 Encapsulation

Encapsulation is a fundamental concept in OOP that combines data and methods into a single unit called a class. Encapsulation allows the class to control access to its data and methods, ensuring that they are used correctly and consistently. This helps to prevent misuse and errors, and promotes good design practices such as information hiding and modularity.

Encapsulation is achieved through the use of access modifiers, which specify the level of access to class members. In Java, access modifiers are enforced by

```python
class BankAccount:
    def __init__(self, balance: float):
        self._balance = balance if balance >= 0 else 0

    def repOK(self):
        return self._balance >= 0

    def deposit(self, amount: float) -> bool:
        """
        REQUIRES: amount must be positive
        EFFECTS: balance is the original balance plus deposited amount
        """
        if amount <= 0:
            return False
        self._balance += amount
        # check_repOK()
        return True

    def withdraw(self, amount: float) -> bool:
        # REQUIRES: amount must be positive and less than or equal to balance
        # EFFECTS: balance is the original balance minus withdrawn amount

        if amount <= 0 or amount > self._balance:
            return False
        self._balance -= amount
        self.check_repOK()
        # check_repOK()
        return True

class BonusBankAccount(BankAccount):
    def __init__(self, balance: float, bonus_interest: float):
        super().__init__(balance)
        self._bonus_interest = bonus_interest

    def deposit(self, amount: float) -> str:
        # REQUIRES: (same) amount must be positive
        # EFFECTS: (stronger) deposit and also add bonus interest

        stats = super().deposit(amount)
        if stats:
            # deposit successful, add interest
            self._balance += self._bonus_interest * amount

        # check_repOK()
        return stats


    def withdraw(self, amount: float) -> bool:
        """
        REQUIRES: (weaker) allow zero withdrawals, which are ignored
        EFFECTS: (same) balance is the original balance minus withdrawn amount
        """
        if amount == 0:
            return True  # Zero withdrawal is considered a no-op
        ret =  super().withdraw(amount)
        # check_repOK()
        return ret


    def repOK(self):
        """
        Stronger Rep-inv: balance and bonus interest must be non-negative
        """
        return super().repOK() and self._bonus_interest >= 0
```

Fig. 4.2: Liskov Substitution Principle demonstration

the language, and there are four levels of access: *private*, *protected*, *package-private* (default), and *public*. In Python, access modifiers are not enforced by the language, but conventions are used to indicate the intended level of access. For example, underscore (_) is used to indicate private or protected attribute (variable).

Encapsulation avoids direct access to the internal representation of a class, e.g., rep-invariants, which can lead to unintended side effects and break the class's invariants. Instead, access to the class's data should be controlled through methods, such as `getters` and `setters` methods.

In the `BankAccount` class in Fig. 4.2, the `_balance` attribute is a private member, and access to it is controlled through the `deposit` and `withdraw` methods. This ensures that the balance is updated correctly and that the rep-invariant is maintained (`repOK`). The `BonusBankAccount` class extends `BankAccount` and adds a `_bonus_interest` attribute, which is also a private member that is not exposed directly.

4.8.1 In-class: Polymorphism concepts: Vehicle

You will design a system that models a fleet of different types of vehicles (e.g., cars, bicycles, trucks). Each vehicle has the ability to start, stop, and display its details. Vehicles should differ in their implementation of these behaviors. You will use abstract classes and interfaces to define the basic structure and ensure that your system adheres to OOP principles.

1. Create an abstract class `Vehicle` that has
   (a) An encapsulated attribute for `speed`.
   (b) Abstract methods: `start()`, `stop()`, and `display()`.
2. Define an interface called `Refuelable`, with a method `refuel(amount:int)`
3. Create concrete subclasses
   (a) Create `Car` and `Bicycle` classes that inherit from `Vehicle`.
   (b) Car also implements the `Refuelable` interface (because it uses fuel).
   (c) Implement methods to `start`, `stop`, `display`, and `refuel` if applicable.
   (d) Ensure each class encapsulates its specific properties (e.g., `fuel_level` for cars).
4. Demonstrate Polymorphism and other OOP principles
   (a) Create a function `operate_vehicle(vehicle:Vehicle)` that accepts any vehicle type and calls its `start`, `stop`, and `display` methods. This function demonstrates polymorphism and dynamic dispatching.

(b) Create test cases to demonstrate LSP by substituting instances of `Car` and `Bicycle` for `Vehicle` in the `operate_vehicle` function.

(c) Protect rep data and other attributes and access them through setters and getters methods.

(d) Provide proper document and specifications for your code (e.g., class Overview, rep-invs, method specifications, AF, `repOK`).

4.9 Exercise

4.9.1 LSP: Market subtype

Determine whether the `LowBidMarket` and `LowOfferMarket` classes are proper subtypes of `Market`. Specifically, for each method in each class, list whether the precondition is weaker, the postcondition is stronger, and conclude whether LSP holds.

Note that this is purely a "paper and pencil" exercise. No code is required. Write your answer so that it is easily understandable by someone with only a passing knowledge of Liskov's rules for subtypes.

```
class Market:
    def __init__(self):
        self.wanted = set()  # items for which prices are of interest
        self.offers = {}     # offers to sell items at specific prices

    def offer(self, item, price):
        """
        Requires: item is an element of wanted.
        Effects: Adds (item, price) to offers.
        """
        if item in self.wanted:
            if item not in self.offers:
                self.offers[item] = []
            self.offers[item].append(price)

    def buy(self, item):
        """
        Requires: item is an element of the domain of offers.
        Effects: Chooses and removes some (arbitrary) pair (item, price) from
                    offers and returns the chosen price.
        """
        if item in self.offers and self.offers[item]:
            return self.offers[item].pop(0)  # Removes and returns the first price
        return None

class LowBidMarket(Market):
    def offer(self, item, price):
        """
        Requires: item is an element of wanted.
        Effects: If (item, price) is not cheaper than any existing pair
                    (item, existing_price) in offers, do nothing.
                    Else add (item, price) to offers.
        """
        if item in self.wanted:
            if item not in self.offers:
```

```
class A:
    def reduce(self, x):
        """
        Effects: if x is None, raise ValueError;
                 if x is not appropriate, raise TypeError;
                 else, reduce this by x.
        """

class B:
    def reduce(self, x):
        """
        Requires: x is not None.
        Effects: if x is not appropriate, raise TypeError;
                 else, reduce this by x.
        """

class C:
    def reduce(self, x):
        """
        Effects: if x is None, return normally with no change;
                 if x is not appropriate, raise TypeError;
                 else, reduce this by x.
        """
```

Fig. 4.3: LSP Exercise

```
                self.offers[item] = []
            # Only add if price is lower than existing prices
            if not self.offers[item] or price < min(self.offers[item]):
                self.offers[item].append(price)

class LowOfferMarket(Market):
    def buy(self, item):
        """
        Requires: item is an element of the domain of offers.
        Effects: Chooses and removes the pair (item, price) with the
                    lowest price from offers and returns the chosen price.
        """
        if item in self.offers and self.offers[item]:
            # Find and remove the lowest price from the list
            lowest_price = min(self.offers[item])
            self.offers[item].remove(lowest_price)
            return lowest_price
        return None
```

4.9.2 LSP: Reducer

For the classes `A`, `B`, and `C` in Fig. 4.3, determine whether LSP holds in the following cases. Specifically, for each case, list whether the precondition is weaker, the postcondition is stronger, and conclude whether LSP holds.

1. `B` extends `A`.
2. `C` extends `A`

3. `A` extends `B`
4. `C` extends `B`
5. `A` extends `C`

4.9.3 LSP Analysis

Consider the following classes with their specifications for the `update()` method:

```
class A:
    def update(self, value):
        """
        Effects/Post: If value is not valid, do nothing;
                otherwise, update this with value.
        """

class B:
    def update(self, value):
        """
        Requires/Pre: value must be an integer.
        Effects/Post: If value is valid, update this with value;
                otherwise, do nothing.
        """

class C:
    def update(self, value):
        """
        Effects/Post: If value is invalid, set default update;
                otherwise, update this with value.
        """
```

For each case below, determine if LSP holds by checking whether the preconditions are weaker and the postconditions are stronger, and conclude whether LSP holds. Note that as soon as one rule is violated, LSP does not hold.

1. `B` extends `A`
2. `C` extends `A`
3. `A` extends `B`
4. `C` extends `B`
5. `A` extends `C`

Chapter 5

Iterators

Iterators and generators are powerful constructs in OOP that enable efficient traversal and on-the-fly computation of sequences of data. They allow developers to handle large datasets, abstract complex data traversal patterns, and create custom iterators for any type of object.

5.1 Motivation

Let's consider a scenario where you need to generate Fibonacci numbers. A common but inefficient approach is to generate all Fibonacci numbers up to a certain limit and store them in a list, which consumes a lot of memory, especially for large sequences.

```
def generate_fib_list(n: int) -> list[int]:
    fib_sequence = [0, 1]
    for _ in range(2, n):
        fib_sequence.append(fib_sequence[-1] + fib_sequence[-2])
    return fib_sequence

# Create the first 100K Fibs; consume lots of memory for storing all numbers
fib_numbers = generate_fib_list(10**6)
print(fib_numbers[:10])  # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34] # only use first 10
```

This approach is inefficient because it generates all Fibonacci numbers up to a certain limit and stores them in a list, which consumes a lot of memory, especially for large sequences. Also, this approach is wasteful because it generates all Fibonacci numbers at once, even if only a few are needed. A more efficient approach is to use an iterator or generator to produce Fibonacci numbers on the fly, only when needed.

```
# Efficient generator function that yields Fibonacci numbers on demand
def fib_generator(n: int):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

print(list(fib_generator(10))) # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Using generator functions, we can efficiently generate Fibonacci numbers on demand, reducing memory consumption and improving performance. Generators produce values one at a time, only when needed, making them ideal for large datasets or infinite sequences.

5.2 Brief History

The concept of iterators in object-oriented programming was pioneered by the language CLU in the 1970s, developed by Barbara Liskov. CLU introduced iterators as a core language feature, allowing traversal of collections without exposing their internal structures, managing the iteration state internally. This innovation laid the foundation for modern iterator designs by demonstrating how encapsulating traversal logic could lead to cleaner, more maintainable code. C++ in the 1980s further popularized iterators through the Standard Template Library (STL), which abstracted the traversal of various collections, enhancing code reuse and flexibility. The importance of iterators was further solidified by the Design Patterns book by the Gang of Four (GoF) in 1994, which formalized the iterator pattern as a key design strategy, emphasizing the separation of traversal from the data structure itself.

Java, released in 1995, built on these ideas by integrating the Iterator interface into its collections framework, standardizing the way collections were traversed across the language. Java's approach unified data traversal, promoting object-oriented principles like encapsulation and abstraction. The introduction of generators in Python in 2001 marked a significant evolution in iteration. Python's yield keyword allowed functions to produce values lazily, one at a time, without storing the entire sequence in memory, enabling efficient data processing for large or infinite sequences. This approach made creating iterators more intuitive and readable, influencing other languages and underscoring the importance of efficient iteration mechanisms in modern programming.

5.3 Iterators

An iterator is an ADT that allows you to traverse through all the elements of a collection, such as a list, tuple, or custom data structure, without exposing the underlying details of the collection (i.e., promoting encapsulation).

Key Concepts of Iterators:

- Iteration Methods: An iterator object implements two key methods: `__iter__()` and `__next__()`.
  - `__iter__()`: Returns the iterator object itself and is implicitly called at the start of loops.
  - `__next__()`: Returns the next element in the sequence and raises a StopIteration exception when there are no more elements.

- State Management: Iterators manage their own state, allowing them to keep track of the current position in the collection.

```
# Creating an iterator for a countdown
class Countdown:
    def __init__(self, start: int):
        self.current = start

    def __iter__(self): return self

    def __next__(self):
        if self.current <= 0:
            raise StopIteration  # End of iteration
        else:
            current_value = self.current
            self.current -= 1
            return current_value

# Usage of the custom iterator
countdown = Countdown(5)
for number in countdown:
    print(number)
# Output: 5, 4, 3, 2, 1
```

In the example above, the `Countdown` class implements iteration by defining the `__iter__()` and `__next__()` methods. The `__iter__()` method returns the iterator object itself, while `__next__()` manages the countdown state by returning the next element in the countdown sequence and stopping the iteration by raising the `StopIteration` exception when the countdown reaches zero.

Benefits of Iterators

- Memory Efficiency: Iterators retrieve elements one at a time, reducing memory usage compared to loading all elements at once.
- Encapsulation: Iterators hide the internal structure of the collection, providing a clean, consistent interface for traversal.
- Flexibility: Custom iterators can be defined for any object, making them adaptable to a wide range of data structures.

5.4 Generator

A generator is a special type of iterator that simplifies the creation of iterators using the `yield` keyword. Generators allow you to declare a method that behaves like an iterator, generating values on the fly without the need to create a separate iterator class or implement `__iter__()` and `__next__()` methods manually.

Generator works by maintaining the state of the function between calls, allowing it to resume execution from where it left off. This lazy evaluation approach is more

memory-efficient than loading all elements at once, making generators ideal for large datasets or infinite sequences.

- Yielding Values: Generators use `yield` to produce a sequence of values one at a time, suspending their state between each yield.
- State Preservation: When a generator function yields, it saves its current state, allowing the function to resume where it left off the next time it is called.
- Lazy Evaluation: Generators compute each value only when needed, making them more efficient for large data sets.

```
# Generator function for a countdown
def countdown(start: int):
    while start > 0:
        yield start
        start -= 1

# Usage of the generator
for number in countdown(5):
    print(number)
# Output: 5, 4, 3, 2, 1
```

Instead of defining the `Countdown` class as an iterator, the `countdown` function is defined as a generator that yields the countdown sequence. The countdown function uses `yield` to generate values one at a time, pausing between each yield. Each call to `yield` returns the current value of start and saves the function'ss state, allowing it to resume where it left off when called again.

Benefits of Generators

- Conciseness: Generators provide a more straightforward syntax for creating iterators, reducing boilerplate code.
- Performance: They generate values on demand, reducing memory consumption compared to traditional lists.
- Enhanced Readability: Generator functions are typically easier to understand and maintain compared to manually implemented iterators.

5.5 In-Class Exercise: Prime Number

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. In this exercise, you will create a generator function that yields prime numbers on demand.

1. Write a non-iterator and non-generator method `gen_prime` that generates prime numbers up to a specified limit.

   (a) Test the iterator by printing all prime numbers that is less than 50.
   (b) Measure the performance of the iterator by generating all prime numbers that your computer can handle (in Python, use `time(...)`). Try various limits and measure the time.

2. Write a custom iterator called `PrimeNumberIterator` that generates prime numbers up to a specified limit.
   (a) The class needs to have `__iter__()` and `__next__()` methods.
   (b) Use a helper function to check for prime numbers (reuse the code in `gen_prime`).
   (c) Raise `StopIteration` when the current number exceeds the limit.
   (d) Test the iterator by printing all prime numbers that is less than 50.
   (e) Measure the performance of the iterator by generating all prime numbers that your computer can handle like before. Try various limits and measure the time.

3. Write a generator function called `gen_prime_generator` that yields prime numbers up to a specified limit (this means using the `yield` keyword).
   (a) Test the generator by printing all prime numbers that is less than 50.
   (b) Measure the performance of the generator by generating all prime numbers that your computer can handle like before. Try various limits and measure the time.

Chapter 6

Testing

Chapter 7

Design Patterns

7.1 Creational Patterns

7.1.1 Singleton

7.1.2 Factory Method

7.2 Structural Patterns

7.3 Behavioral Patterns

7.4 Composition over Inheritance

Appendix A

Miscs

Appendix B

More Examples

B.1 ADT

B.1.1 Stack ADT

```
class Stack:
"""
Overview: Stack is a mutable ADT that represents a collection of elements in LIFO.
AF(c) = the sequence of elements in the stack in sorted order from bottom to top.
rep-inv:
    1. elements is a list (could be empty list, which represents and empty stack).
    2. The top of the stack is always the last element in the list.
"""

def __init__(self):
    """
    Constructor
    EFFECTS: Initializes an empty stack.
    MODIFIES: self
    """
    self.elements = []

def repOK(self):
    """
    EFFECTS: Returns True if the rep-invariant holds, otherwise False.
    The invariant checks:
    1. elements is a list.
    2. If the stack is non-empty, the top of the stack is the last element in the list.
    """
    # Check that elements is a list
    if not isinstance(self.elements, list):
        return False

    # If the stack is not empty, ensure that the top is the last element in the list.
    # This is implicitly guaranteed by the use of `list.append` for push and `list.pop` for
    # so no further explicit check is needed for the "top as last element."
    return True

def push(self, value):
    """
    MODIFIES: self
    EFFECTS: Adds value to the top of the stack.
    """
    self.elements.append(value)

def pop(self):
    """
    MODIFIES: self
    EFFECTS: Removes and returns the top element from the stack.
    Raises an exception if the stack is empty.
    """
    if self.is_empty():
        raise Exception("Stack is empty")
    return self.elements.pop()

def is_empty(self):
    """
    EFFECTS: Returns True if the stack is empty, otherwise False.
    """
    return len(self.elements) == 0

def __str__(self):
    """
    EFFECTS: Returns a string representation of the stack,
             showing the elements from bottom to top.
    """
    # The abstraction function maps the list of elements to a stack view
    return f"Stack({self.elements})"
```

Fig. B.1: Stack ADT