SUPSI

Abstract Data Types

Introduction to Algorithms and Data Structures

Bachelor in Data Science and Artificial Intelligence

Types in Python: primitive and user defined

- **Primitive data types** are part of the programming language
- **User defined data types** are defined by the programmer, not by the language, and they can be very complex

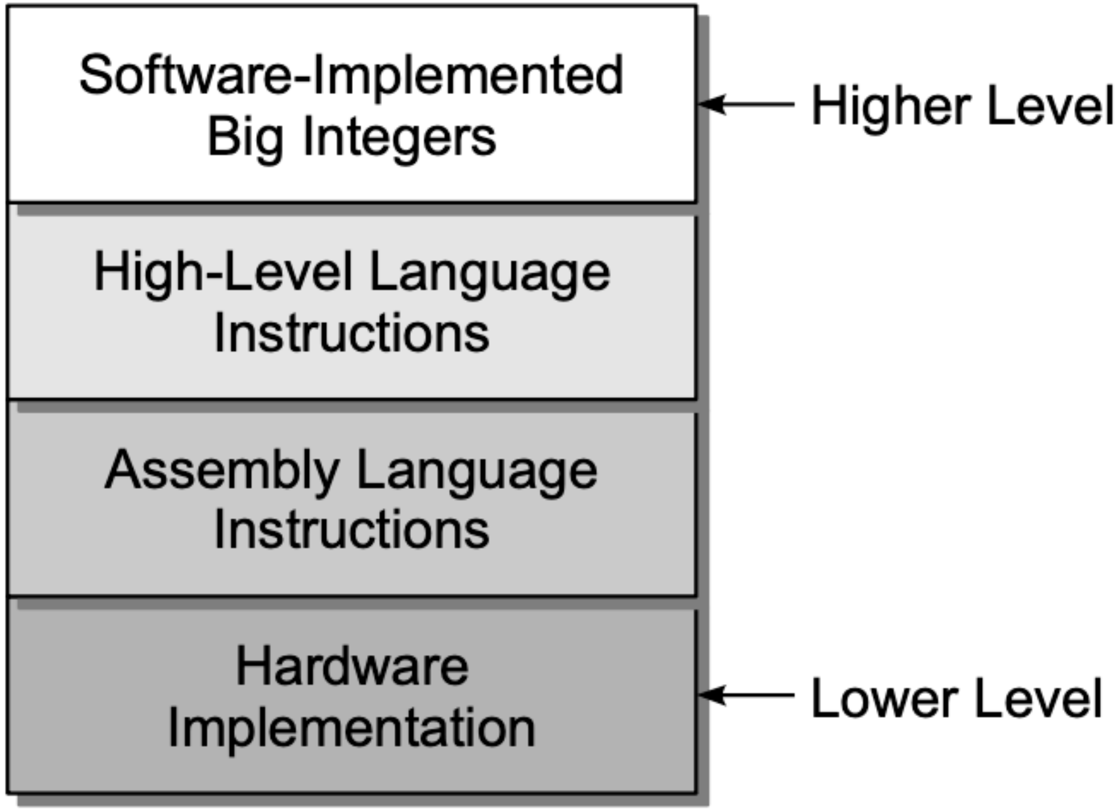
Type	Description	Examples
Primitive (Simple)	Cannot be decomposed further	`int`, `float`, `bool`
Primitive (Complex)	Built using simple types	`str`, `list`, `dict`
User-defined	Defined by the programmer	Classes, Objects

Abstractions

- **Functional Abstraction**: Functions encapsulate logic, and we use them without knowing the details.
  - Example: sqrt(x) calculates the square root without requiring us to understand its implementation.
- **Data Abstraction**: Separates the properties of a data type from its implementation.
  - Example: Strings in Python, which internally use arrays but abstract complexity away.
- Even arithmetic operations are abstractions: $x = a + b - 5$ is an abstraction of

```
loadFromMem( R1, 'a' )
loadFromMem( R2, 'b' )
add R0, R1, R2
sub R0, R0, 5
storeToMem( R0, 'x' )
```

Example of abstraction: integers

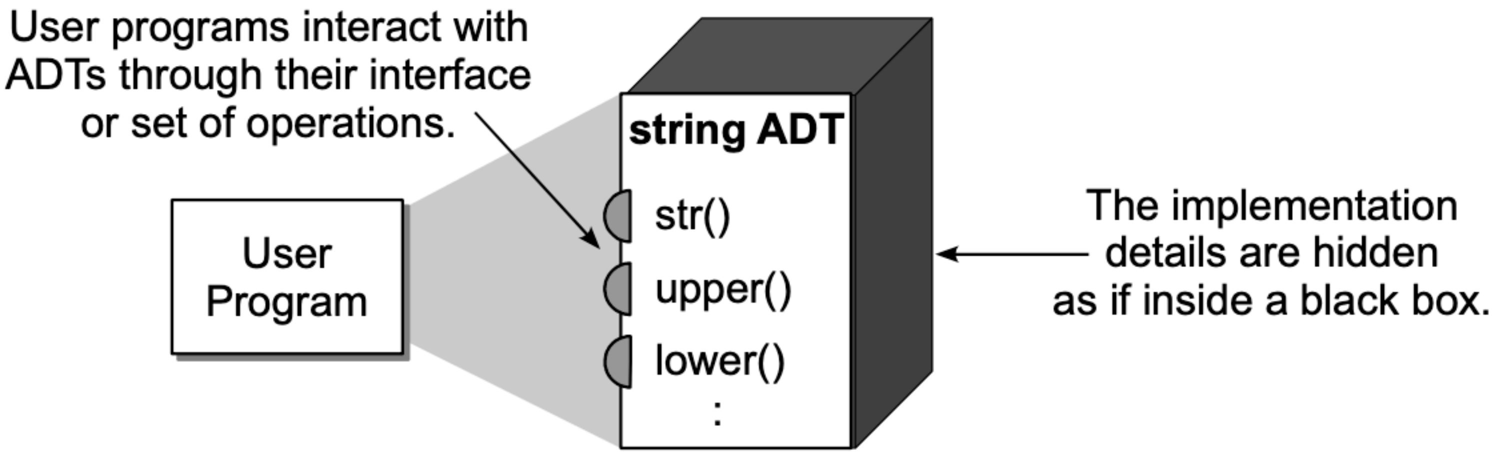


Python provides software implementations of big integers as part of the language

Abstract data types

- It is a **data type**, defined by the programmer
- It specifies a set of **data values**
- It specifies a collection of **operations** that can be performed on the data values

- The **interface** is separate from the **implementation** => **information hiding**




```
Operations can be:
  ◦ Constructors
  ◦ Accessors
  ◦ Mutators
  ◦ Iterators
```

Some advantages of ADT

- **Problem-focused:** Avoids dealing with implementation details.
- **Prevents logical errors:** Restricts direct access to implementation.
- **Modifiability:** Changes to implementation **do not affect the interface**.
- **Supports modular programming** (*divide-et-impera*).

Data structures

Simple ADT

```python
class Date:
    def __init__ (self, day, month, year):
        self.day=day
        self.month=month
        self.year=year
```

Complex ADT

```python
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items) - 1]

    def size(self):
        return len(self.items)
```

Data structures: general definitions

- **Collection**: no relationship or ordering among the values
- **Container**: any data structure or ADT that stores and organises a collection
  - The individual values of the collection are **elements** of the container
- **Sequence**: a container where the elements are arranged in linear order and are accessible by position
- **Sorted sequence**: the position of the elements in the sequence is based on a prescribed relationship
- **List** is a collection with linear ordering: each element, except the first one, has a unique successor. A sequence is a list, but a list is not necessarily a sequence.
- The Python list is actually better called an array list, as in Java, but we will call it *list* and we will call the “list” *general list*

The Date ADT

Defining the Date ADT

- 1. Define the domain of the data elements
- 2. Define the set of operations

- The Gregorian calendar was introduced in 1582 by Pope Gregorius XIII to account for the errors in the Julian calendar and introduced the leap year

- The first date in the Gregorian calendar is October 15, 1582, AD

- The proleptic Gregorian calendar is an extension that allows for earlier dates, starting on the 1st of January 4713 BC, that is year 1 in the Julian calendar period (https://en.wikipedia.org/wiki/Julian_day)

- The proleptic Gregorian calendar facilitates the handling of dates across calendars and it is found in many software packages

Defining the Date ADT: named methods

- `Date(month, day, year)`: creates a new `Date` instance initialised to the given Gregorian date which must be valid. Year 1 BC and earlier are indicated by negative year components.
- `day()`: Returns the Gregorian day number of this date.
- `month()`: Returns the Gregorian month number of this date.
- `year()`: Returns the Gregorian year of this date.
- `monthName()`: Returns the Gregorian month name of this date.
- `dayOfWeek()`: Returns the day of the week as a number between 0 and 6 with 0 representing Monday and 6 representing Sunday.
- `numDays(otherDate)`: Returns the number of days as a positive integer be- tween this date and the otherDate.
- `isLeapYear()`: Determines if this date falls in a leap year and returns the appropriate boolean value.
- `shiftBy(days)`: Shifts the date by the given number of days. The date is incremented if days is positive and decremented if days is negative. The earliest date is limited to 1st of January, 4713 BC.

Defining the Date ADT: operators

- Python allows classes to **overload** operators. We implement two operators
- *`comparable`* `(otherDate)`: Compares this date to the otherDate to determine their logical ordering. This comparison can be done using any of the logical operators <, <=, >, >=, ==, !=.
- *`toString ():`* Returns a string representing the Gregorian date in the format mm/dd/yyyy. Implemented as the Python operator that is automatically called via the **`str()`** constructor.

```
class Date:
    def __lt__(self, other):
        return (self.year,
self.month, self.day) < (other.year,
other.month, other.day)
```

```
d1 = Date(10, 5, 2024)
d2 = Date(11, 5, 2024)
print(d1 < d2)  # True
```

Using the Date ADT

```python
# Extracts a collection of birth dates from the user and determines
# if each individual is at least 21 years of age.
from date import Date

def main():
    # Date before which a person must have been born to be 21 or older.
    bornBefore = Date(10, 1, 1999)

    # Extract birth dates from the user and determine if 21 or older.
    date = promptAndExtractDate()
    while date is not None :
        if date <= bornBefore :
            print( "Is at least 21 years of age: ", date )
        else:
            print("Sorry, not of age: ",  date)
        date = promptAndExtractDate()
```

Using the Date ADT

```
# Prompts for and extracts the Gregorian date components.
# Returns a Date object or None when the user has finished entering dates.

def promptAndExtractDate():
    print( "Enter a birth date." )
    month = int( input("month (0 to quit): ") )
    if month==0:
        return None
    else :
        day = int( input("day: ") )
        year = int( input("year: ") )
        return Date( month, day, year )

# Call the main routine.
main()
```

Preconditions and postconditions

- A **precondition** indicates the condition or state of the ADT instance and inputs before the operation can be performed.
- A **postcondition** indicates the result or ending state of the ADT instance after the operation is performed

- All operations have at least one precondition, which is that the ADT instance has to have been previously initialised.
- Some operations may not have a postcondition, as is the case for simple access methods, which simply return a value without modifying the ADT instance itself.

- When a pre or post-condition is not satisfied an **exception** is raised

- The `assert` statement can be used to test the preconditions

Implementing the Date ADT

The date is stored as a Julian value
Which simplifies the comparison operations

```
class Date :
    # Creates an object instance for the specified Gregorian date.
    def __init__( self, month, day, year ):
        self._julianDay = 0
        assert self._isValidGregorian( month, day, year ), \
          "Invalid Gregorian date."
        # The first line of the equation, T = (M - 14) / 12, has to be changed
        # since Python's implementation of integer division is not the same
        # as the mathematical definition.
        tmp = 0
        if month < 3 :
            tmp = -1
        self._julianDay = day - 32075 + \
            (1461 * (year + 4800 + tmp) // 4) + \
            (367 * (month - 2 - tmp * 12) // 12) - \
            (3 * ((year + 4900 + tmp) // 100) // 4)
```

Protected Attributes and Methods.

Python does not provide a technique to protect attributes and helper methods in order to prevent their use outside the class definition. In this text, we use identifier names, which begin with a single underscore to flag those attributes and methods that should be considered protected and rely on the user of the class to not attempt a direct access.

Implementing the Date ADT

```
# Returns day of the week as an int between 0
(Mon) and 6 (Sun).

def dayOfWeek( self ):
    month, day, year = self._toGregorian()
    if month < 3 :
        month = month + 12
        year =year-1
    return ((13*month+3)//5+day+\
            year+year//4-year//100+year//400)%7
```

```
# Logically compares the two dates.

def __eq__( self, otherDate ):
    return self._julianDay == otherDate._julianDay

def __lt__( self, otherDate ):
    return self._julianDay < otherDate._julianDay

def __le__( self, otherDate ):
    return self._julianDay <= otherDate._julianDay
```

By implementing the methods for the logical comparison operators, instances of the class become comparable objects.

Implementing the Date ADT

```
# Returns the Gregorian date as a
tuple: (month, day, year).

def _toGregorian( self ):
    A = self._julianDay + 68569
    B = 4 * A // 146097
    A = A - (146097 * B + 3) // 4
    year = 4000 * (A + 1) // 1461001
    A= A - (1461*year// 4) +31
    month = 80 * A // 2447
    day=A-(2447 * month// 80)
    A=month // 11
    month = month + 2 - (12 *A)
    year=100*(B-49) + year +A
    return month, day, year
```

Implementing the Date ADT

```
# Extracts the appropriate Gregorian date component.
  def month( self ):
      return (self._toGregorian())[0] # returning M from (M, d, y)

  def day( self ):
      return (self._toGregorian())[1] # returning D from (m, D, y)

  def year( self ):
      return (self._toGregorian())[2] # returning Y from (m, d, Y)
```

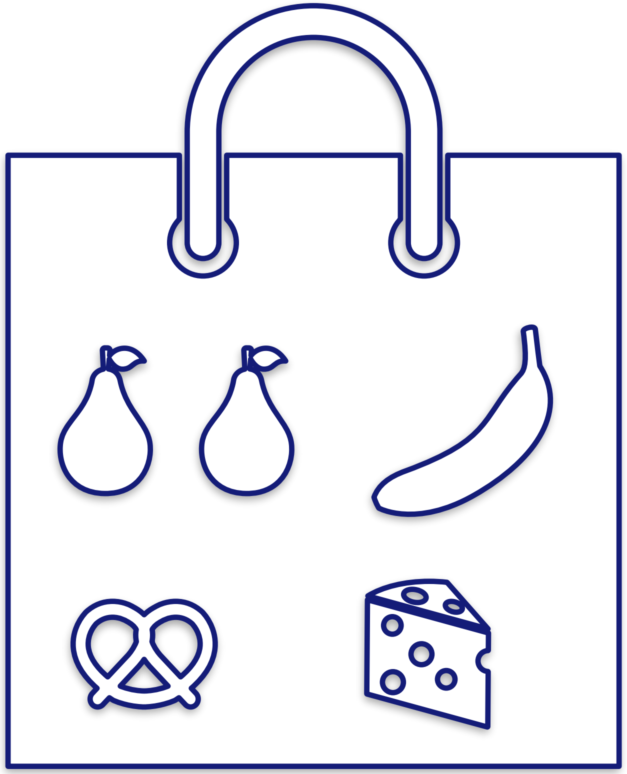
Hands-on activity (25 min)

- Take a look at the code of the Date class presented in the textbook and implement it in two Python files
    - main.py - containing the main body that will use the implemented functions
        - In this file you should test that 2024 is actually a leap year
        - Use it to enter your DoB and return the day of the week of your birth date
    - date.py - containing the example provided
- Is important that you read the code that has been provided and you understand its content.
- BEWARE: there may be a few mistakes in the code of the book, fix them!

Bags

The Bag ADT

- The Date ADT was a simple ADT: no nesting of types, just primitives
- The Bag ADT is a **complex** ADT
- It is a simple container like a shopping bag
  - It stores a collection of items
  - There might be duplicates
  - The operations are limited
    - I can add an item
    - I can remove an item
    - I can check if an item is in the bag
    - I can scan the content of the bag



Definition of the Bag ADT

- `Bag()`: Creates a bag that is initially empty.
- `length()`: Returns the number of items stored in the bag. Accessed using the **`len()`** function.
- `contains (item)`: Determines if the given target item is stored in the bag and returns the appropriate boolean value. Accessed using the **`in`** operator.
- `add(item)`: Adds the given item to the bag.
- `remove(item)`: Removes and returns an occurrence of item from the bag. An exception is raised if the element is not in the bag.
- `__iter__()`: Creates and returns an *iterator* that can be used to iterate over the collection of items.

Example of use of the Bag ADT

```
myBag = Bag()
myBag.add( 19 )
myBag.add( 74 )
myBag.add( 23 )
myBag.add( 19 )
myBag.add( 12 )

value = int( input("Guess a value contained in the bag.") )
if value in myBag:
   print( "The bag contains the value", value )
else:
   print( "The bag does not contain the value", value )
```

Selecting a data structure

- How to select?
  - Does the data structure provide for the storage requirements as specified by the domain of the ADT?
    - (e.g. can an array store objects?)
  - Does the data structure provide the necessary data access and manipulation functionality to fully implement the ADT?
    - (e.g. does an array provide indexed access to elements? Can I use the functionality without breaking up the encapsulation rules?)
  - Does the data structure lend itself to an efficient implementation of the operations?
    - (e.g. does a list performs efficiently for random access of its elements)

Selecting a data structure for the Bag ADT

- Possible candidates: list vs dictionary
  - The list stores any comparable object including duplicates
  - The dictionary stores key/values pairs, and the keys must be unique. We could add a counter for the number of duplicates we have, while maintaining just one instance
  - If most items are unique, the list is more size efficient

Implementing the Bag operations with the list

- An empty bag can be represented by an empty list.
- The size of the bag can be determined by the size of the list.
- Determining if the bag contains a specific item can be done using the equivalent list operation.
- When a new item is added to the bag, it can be appended to the end of the list since there is no specific ordering of the items in a bag.
- Removing an item from the bag can also be handled by the equivalent list operation.
- The items in a list can be traversed using a for loop and Python provides for user-defined *iterators* that be used with a bag.

The linearbag.py module

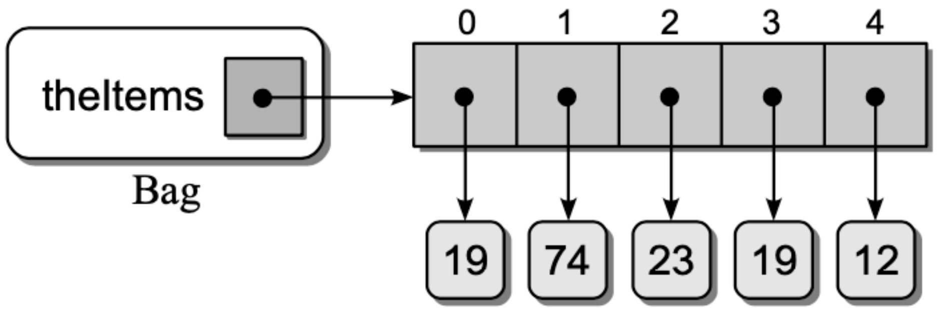
```
# Implements the Bag ADT container using a Python list.
class Bag:

    # Constructs an empty bag.
    def __init__(self):
        self._theItems = list()

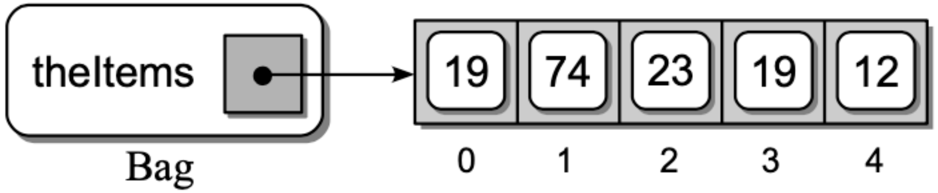
    # Returns the number of items in the bag.
    def __len__(self):
        return len(self._theItems)

    # Determines if an item is contained in the bag.
    def __contains__(self, item):
        return item in self._theItems
```

The correct representation



A more compact representation



The linearbag.py module

```
# Adds a new item to the bag.
    def add(self, item):
        self._theItems.append(item)

    # Removes and returns an instance of the item from the bag.
    def remove(self, item):
        assert item in self._theItems, "The item must be in the bag."
        ndx = self._theItems.index(item)
        return self._theItems.pop(ndx)

    # Returns an iterator for traversing the list of items.
    def __iter__(self, item):
```

Iterators

Traversing a container: the iterator is the answer

- Python’s containers (strings, tuples, lists, and dictionaries) can be traversed with the for loop
- This is because we know the structure of such containers is known
- What if the type is user defined, and the structure is unknown?
- The user must also define “how to” traverse the data type: enter **the iterator**

- The iterator allows to preserve the principle of abstraction: the internal data structure of the ADT is not accessed

- The iterator guarantees flexibility of use: we are not constrained to a “canned” use of the ADT (e.g. a “print” method supplied by the ADT would be limiting)

- Python, like many of today’s object-oriented languages, provides a built-in iterator construct that can be used to perform traversals on user-defined ADTs. An iterator is an object that provides a mechanism for performing generic traversals through a container without having to expose the underlying implementation

Design an iterator

- We must create an iterator class: a Python class with at least two methods `__iter__` and `__next__`
- We define the class `BagIterator`
- The constructor defines two data fields.
  - an alias to the list used to store the items in the bag,
  - a loop index variable that will be used to iterate over that list.
- The `__iter__` method simply returns a reference to the object itself
- The `__next__` is called to return the next item in the container
- Finally an `__iter__` method must be added to the Bag

  - ```
    def __iter__( self ):
        return _BagIterator( self._theItems )
    ```
    ```

The BagIterator implementation

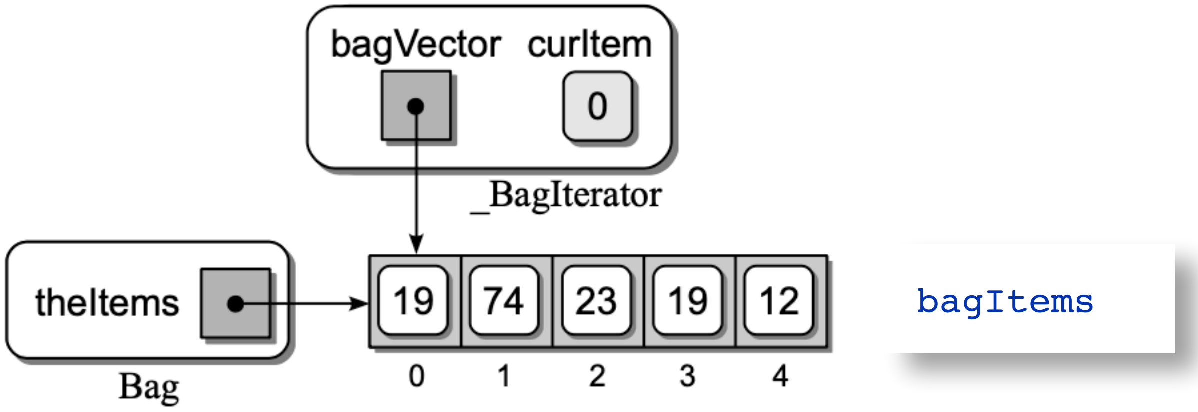
```python
class _BagIterator:

    # Constructs a BagIterator
    def __init__(self, theList):
        self._bagItems = theList
        self._curItem = 0

    # Returns self
    def __iter__(self):
        return self

    # Return the next element
    def __next__(self):
        if self._curItem < len(self._bagItems):
            item = self._bagItems[self._curItem]
            self._curItem += 1
            return item
        else:
            raise StopIteration
```

How to use an iterator

- It is the simplest thing:

```
for item in MyBag:
    print(item)
```

- Python calls the __iter()__ method on the bag to create a _BagIterator



Hands-on activity (25 min)

- Take a look at the code of the Bag class presented in the textbook and implement it in two Python files
    - main.py - containing the main body that will use the implemented functions
    - linearbag.py - containing the example provided
- Is important that you read the code that has been provided and you understand its content.
- BEWARE: there may be a few mistakes in the code of the book, fix them!