

# CONTAINERS, DATA ABSTRACTION, AND SEQUENCES

---

COMPUTER SCIENCE MENTORS 61A

October 3–October 007, 2022

---

## 1 Lists

---

### Lists Introduction:

Lists are a data structure, an ordered collection of values that contain any type of data, even more lists itself! This may be a new concept for you if you have background in other languages, like Java, where lists can only be made of the same type elements.

In order to write a list, we wrap our elements with square brackets, and separate elements with commas.

We can access specific properties of a list, such as the length, and the specific index of an item in the list, although it is important to note that when we index into lists, we start at 0, and not 1.

```
>>> lst = [1, False, [2, 3], 4] # a list can contain any data
      type
>>> len(lst)
4
>>> lst[0] # Indexing starts at 0
1
>>> type(lst[2]) # Can have lists within lists!
<class 'list'>
>>> lst[4] # Indexing ends at len(lst) - 1
Error: list index out of range
```

We can iterate over lists using their index, or iterate over elements directly

```
a = [1, 2, 3, 4]
```

```

for index in range(len(lst)):
    print(a[index])
for item in lst:
    print(item)

```

Both for loops will output:

```
1 2 3 4
```

**List comprehensions** are a concise and powerful method to create a new list out of sequences. The general syntax of list comprehensions follows:

```
[<expression> for <element> in <sequence> if <condition>]
```

The **if** <condition> is optional, and an equivalent for loop for a list comprehension is as follows:

```

lst = []
for <element> in <sequence>:
    if <condition>:
        lst = lst + [<expression>] # add current
                                element to the list

```

An example of list comprehensions in use:

```

>>> [i * 2 for i in [1, 2, 3, 4] if i % 2 != 0] # iterate over
      numbers from [1, 2, 3, 4] and if they are odd, multiply them
      by 2
[2, 6]

```

Equivalence in for loop syntax:

```

lst = []
for i in [1, 2, 3, 4]:
    if i % 2 != 0:
        lst = lst + [i * 2] # add current
                            element to the list

```

We can use **list slicing** to create a copy of a certain portion or all of a list.

The general syntax for slicing a list `lst` is as follows:

```
lst[<start index>:<end index>:<step size>]
```

Where the portion of the list we want to keep starts at <start index> and ends one before <end index>. Once we have that portion of the list, if we want to only keep some elements, we can add the optional parameter <step size>, which will allow us skip or reverse those elements.

If we do not supply <start index> or <end index>, the default parameter, will be the beginning, and the end of the list, respectively.

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[2:] # Create a new list with only the elements starting
           from the 2nd index
[3, 4, 5]
>>> lst[:3] # Create a new list with only the elements up until
           the 4th index
[1, 2, 3]
>>> lst[::-1] # Reverse the entire list
[5, 4, 3, 2, 1]
>>> lst[::2] # Create a new list while skipping every other
           element
[1, 3, 5]
```

1. What would Python display? Draw box-and-pointer diagrams for the following:

```
>>> a = [1, 2, 3]
>>> a
```

```
>>> a[2]
```

```
>>> a[-1]
```

```
>>> b = a
>>> a = a + [4, [5, 6]]
>>> a
```

```
>>> b
```

```
>>> c = a
>>> a = [4, 5]
>>> a
```

```
>>> c
```

```
>>> d = c[3:5]
>>> c[3] = 9
>>> d

>>> c[4][0] = 7
>>> d

>>> c[4] = 10
>>> d

>>> c
```

2. Draw the environment diagram that results from running the code.

```
def reverse(lst):
    if len(lst) <= 1:
        return lst
    return reverse(lst[1:]) + [lst[0]]

lst = [1, [2, 3], 4]
rev = reverse(lst)
```

3. Write a function that takes in a list `nums` and returns a new list with only the primes from `nums`. Assume that `is_prime(n)` is defined. You may use a **while** loop, a **for** loop, or a list comprehension.

```
def all_primes(nums):
```

4. Write a list comprehension that accomplishes each of the following tasks.

(a) Square all the elements of a given list, `lst`.

(b) Compute the dot product of two lists `lst1` and `lst2`. *Hint:* The dot product is defined as  $lst1[0] \cdot lst2[0] + lst1[1] \cdot lst2[1] + \dots + lst1[n] \cdot lst2[n]$ . The Python **zip** function may be useful here.

(c) Return a list of lists such that `a = [[0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4]]`.

(d) Return the same list as above, except now excluding every instance of the number 2: `b = [[0], [0, 1], [0, 1], [0, 1, 3], [0, 1, 3, 4]]`.



#### Data Abstraction Overview:

Abstraction allows us to create and access data through a controlled, restricted programming interface, hiding implementation details and encouraging programmers to focus on how data is used, rather than how data is organized. The two fundamental components of a programming interface are a constructor and selectors.

1. Constructor: The interface that creates a piece of data; e.g. calling `c = car("Tesla")` creates a new car object and assigns it to the variable `c`.
2. Selectors: The interface by which we access attributes of a piece of data; e.g. calling `get_brand(c)` should return `"Tesla"`.

Through constructors and selectors, abstract data can hide its implementation, and a programmer doesn't need to *know* its implementation to *use* it.

1. The following is abstract data for representing Pokemon. Each Pokemon keeps track of its name, age, type, ability to attack, and friends. Given our provided constructor, fill out the selectors:

```
def pokemon(name, generation, type, can_attack, friends):  
    """  
    Takes in a string name, an int generation, a string  
        type, a boolean can_attack, and a list friends.  
    Constructs an Pokemon with these attributes.  
    >>> cyndaquil = pokemon("Cyndaquil", 2, "Fire", True,  
        ["Chikorita", "Totodile"])  
    >>> pokemon_name(cyndaquil)  
    "Cyndaquil"  
    >>> pokemon_age(cyndaquil)  
    10  
    >>> pokemon_type(cyndaquil)  
    "Fire"  
    >>> pokemon_can_attack(cyndaquil)  
    True  
    >>> pokemon_friends(cyndaquil)  
    ["Chikorita", "Totodile"]  
    """  
    return [name, age, type, can_attack, friends]
```



```
def pokemon_name(p) :
```

```
def pokemon_age(p) :
```

```
def pokemon_friends(p) :
```

2. This function returns the correct result, but there's something wrong about its implementation. Why is the error an error in the first place, and how can we fix it?

```
def pokemon_team(pokemen) :  
    """  
    Takes in a list of Pokemon (aptly named Pokemen due  
    to the plural of Pokemon being Pokemon) and  
    returns a list of their names.  
    """  
    return [pokemon[0] for pokemon in pokemen]
```

3. Fill out the following constructor for the given selectors.

```
def pokemon(name, generation, type, can_attack, friends):
```

```
    def pokemon_generation(p):  
        return e[0][1]
```

```
    def pokemon_can_attack(p):  
        return e[1][0]
```

```
    def pokemon_friends(p):  
        return e[2]
```

4. How can we write the fixed `pokemon_roster` function for the constructors and selectors in the previous question?

5. (Optional) Fill out the following constructor for the given selectors.

```
def pokemon(name, generation, type, can_attack, friends):  
    """  
    >>> lil_guy = pokemon("Pikachu", 1, "Electric",  
        False, ["Mewtwo", "Lucario"])  
    >>> pokemon_name(lil_guy)  
    "Pikachu"  
    >>> pokemon_generation(lil_guy)  
    1  
    >>> pokemon_type(lil_guy)  
    "Electric"  
    >>> pokemon_can_attack(lil_guy)  
    False  
    >>> pokemon_friends(lil_guy)  
    ["Mewtwo", "Lucario"]  
    >> lil_guy("type")  
    "Breaking abstraction barrier!"  
    """  
    def select(command):
```

```
        return select
```

```
def pokemon_name(p):  
    return p("name")
```

```
def pokemon_generation(p):  
    return p("generation")
```

```
def pokemon_type(p):  
    return p("type")
```

```
def pokemon_can_attack(p):  
    return p("can_attack")
```

```
def pokemon_friends(p):
```

```
    return p("friends")
```

In the following problem, we will represent a bookshelf object using dictionaries.

In the first section, we will set up the format. Here, we will directly work with the internals of the Bookshelf, so don't worry about abstraction barriers for now. Fill in the following functions based on their descriptions (the constructor is given to you):

```
def Bookshelf(capacity):
    """ Creates an empty bookshelf with a certain max
        capacity. """
    return {'size': capacity, 'books': {}}

def add_book(bookshelf, author, title):
    """
    Adds a book to the bookshelf. If the bookshelf is
    full,
    print "Bookshelf is full!" and do not add the book.
    >>> books = Bookshelf(2)
    >>> add_book(books, 'Jane Austen', 'Pride and
        Prejudice')
    >>> add_book(books, 'Daniel Kleppner', 'An
        Introduction to Mechanics 5th Edition')
    >>> add_book(books, 'Kurt Vonnegut', 'Galapagos')
    Bookshelf is full!
    """
    if _____:
        print('Bookshelf is full!')
    else:
        if author in bookshelf['books']:
            _____
        else:
            _____
```

```

def get_all_authors(bookshelf):
    """
    Returns a list of all authors who have at least one
    book in the bookshelf.
    >>> books = Bookshelf(10)
    >>> add_book(books, 'Jane Austen', 'Pride and
        Prejudice')
    >>> add_book(books, 'Sheldon Axler', 'Linear Algebra
        Done Right')
    >>> add_book(books, 'Kurt Vonnegut', 'Galapagos')
    >>> get_all_authors(books)
    ['Jane Austen', 'Sheldon Axler', 'Kurt Vonnegut']
    """
    return _____

```

```

def get_author_books(bookshelf, author):
    """
    Given an author name, returns a list with
    all books on the bookshelf written by that author.
    >>> books = Bookshelf(100)
    >>> add_book(books, 'Orson Scott Card', "Ender's
        Game")
    >>> add_book(books, 'Orson Scott Card', 'Speaker for
        the Dead')
    >>> add_book(books, 'J.R.R. Tolkien', 'The Hobbit')
    >>> get_author_books(books, 'Orson Scott Card')
    ['Ender's Game', 'Speaker for the Dead']
    """
    return _____

```

Now, complete the function `most_popular_author` **without breaking the abstraction barrier**. In other words, you are not allowed to assume anything about the implementation of a `Bookshelf` object, or use the fact that it is a dictionary. You can only use the methods above and their stated return values.

```
def most_popular_author(bookshelf):
    """
    Returns the author with the greatest number of books
    on this bookshelf.
    You can assume that the bookshelf is not empty.
    >>> books = Bookshelf(100)
    >>> add_book(books, 'Orson Scott Card', 'Xenocide')
    >>> add_book(books, 'Orson Scott Card', 'Children of
    the Mind')
    >>> add_book(books, 'J.R.R. Tolkien', 'The Hobbit')
    >>> most_popular_author(bookshelf)
    'Orson Scott Card'
    """
    return
    max(_____,
    key=_____)
```

1. Write a function `count_t`, which takes in a dictionary **dict** and a string `word`. The function should count the instances of the letter "t" in `word` and add a key-value pair to the dictionary. The key will be `word` and the value will be the number of "t"s in `word`

```
def count_t(d, word):  
    """  
    >>> words = {}  
    >>> count_t(words, "tatter")  
    >>> words["tatter"]  
    3  
    >>> count_t(words, "tree")  
    >>> words  
    {'tatter': 3, 'tree': 1}  
    """
```

```
    _____  
  
    for _____:  
        if _____:  
            _____  
    _____
```