# CSC110 Tutorial 10: Abstract Data Types and Inheritance

Print this handout

In this tutorial, you'll review some of the new concepts we've covered this week: various abstract data types (both their definition and implementations), and inheritance in Python.

## Exercise 1: Selecting the right abstract data type

We have now covered a variety of collection-based abstract data types in this course:

- Set
- List
- Mapping
- Stack
- Queue
- Priority Queue

An important skill to practice is to take a problem description and determine what the necessary abstract data type(s) are for representing this problem in a Python program. Here are some practice scenarios; for each one identify which of these ADTs you would use to represent the problem. Each ADT may be used more than once; each problem description may require more than one ADT, or an ADT that contains instances of another ADT.

1. An online ticket system for a busy show (like Hamilton) has an online lobby where people can join starting on a particular date. Because of limited server capabilities, users are allowed to select their tickets one at a time, going in the order that they joined the lobby.
2. Amy is stressed about all the upcoming assignments that are due in the next two weeks and to manage her time better, she decides to write them down in a TODO list. She realizes that some work is due sooner than others, so she mentally sorts them by priority to get it done one assignment at a time.
3. Evan plans a party for his 20th birthday where he invites his friends Amy, Callum, David, and Mario. They all drive up to his place separately. When they arrive, they realized that Evan's driveway is very long, but only one car wide. As consequence, whoever arrives first has to be the last to leave.
4. Callum decides to film a video of himself showing off all the skateboarding tricks he learned during quarantine. When editing the video, he makes sure to cut out any unsuccessful attempts and only keep the best version of each trick.
5. During course enrollment, a list of courses is available on ACORN. In the past, the space in popular courses fill up quick. As a result, a waitlist on a first-come first-serve basis is created for each course.
6. Required textbooks for courses can be found in the UofT bookstore. They usually sit on different shelves, separated by major. For example, in order to find "Programming for Smarties", you have to go down the aisle labeled "Computer Science", and walk past piles of books such as "Robot Psychology" and "Trying Stuff until it Works".

## Exercise 2: Implementing the Mapping ADT using `list`

In Section 10.4 of the Course Notes, we discussed the Mapping ADT, and said that we could implement this abstract data type in Python using either `dict` or `list`. In this exercise, you'll explore how to implement this ADT using a `list`.

To start, please download the starter file `tutorial10_part2.py`.

1. First, we'll need to define a new class that has a single `list` instance attribute.

   ```
   class ListMap:
       """An implementation of the Mapping ADT using a list."""
       # Private Instance Attributes:
       #   - _pairs: A list of tuples, where each tuple stores a (key, value) pair
       #             in the mapping.
       _pairs: list[tuple[Any, Any]]
   ```

   Suppose we have a Mapping that contains the following key-value pairs:

   | Key | Value |
   |---|---|
   | `'a'` | `1` |
   | `'b'` | `'David'` |
   | `'c'` | `[1,2,3]` |

   If we have a `ListMap` instance that represents this Mapping, what would its `_pairs` attribute be?

2. Open `tutorial10_part2.py`. Implement the initializer for `ListMap`, which takes in no arguments other than `self`, and initializes an empty Mapping.

3. Implement each of the following Mapping operations as methods for this class.

   - Get size
   - Key assignment (assign a given key to a given value in the mapping; overwrites if the key is already in the mapping)
   - Key lookup
   - Key deletion

   See the starter file for details.

4. Analyse the running time of each of the four operations you implemented in Question 3. Where appropriate, analyse the *worst-case running time* (when the running time is not a function of the size of mapping `self`). Make sure to practice finding upper and lower bounds on the worst-case separately!

   How do these compare to the running times of the corresponding `dict` operations?

## Exercise 3: Inheritance with game strategies

In this section you'll write code to play a simple number game. This game is played with two players. When the game starts, there is a count that begins at 0. On a player's turn, they add to the count an integer that must be between a set minimum and a set maximum. The player whose move causes the count to be greater than or equal to a set goal amount is the winner.

Here's a sample game with two players, where the goal is 21, the minimum move is 1, and the maximum move is 3. David is the winner.

| Mario | David | count |
|---|---|---|
| | | 0 |
| 2 | | 2 |
| | 3 | 5 |
| 3 | | 8 |
| | 1 | 9 |
| 3 | | 12 |
| | 3 | 15 |
| 1 | | 16 |
| | 1 | 17 |
| 3 | | 20 |
| | 1 | **21** |

### 1. Review class `NumberGame`

Download the starter file `tutorial10_part3.py`.

Read the `NumberGame` class carefully and answer the following questions about it. Note that the entire class is provided for you, and your job here is to understand it—in other words, you're practicing your code *reading* skills.

1. What attribute stores the players of the game?
2. If `turn` is 15, whose turn is it?
3. Write a line of code that would create an instance of `NumberGame` that violates one of the representation invariants.

   (`python_ta.contracts` should raise an error when you try to evaluate your expression in the Python console.)

4. List all the places in this class where a `Player` is stored, an instance attribute of `Player` is accessed or assigned to, or a `Player` method is called.

### 2. Designing classes

Since you have found all the places where a `Player` is used, you know the attributes and methods it must provide as its public interface. You could complete the program by writing a single class `Player` with methods that provide these. But we're going to have two different kinds of player (and add a third one in the Additional Exercises). They will have some things in common, but they will differ on how they choose a move:

- A **random AI player** will pick a random move from among the allowed possibilities.
- A **human player** will prompt the user to select a move rather than having the computer choose the move.

Rather than make three unrelated classes, we are going to define a parent class called `Player` and make two child classes.

1. Your three classes are going to be called `Player`, `RandomPlayer`, and `UserPlayer`. Write headers and short docstrings for each of these classes in `tutorial10_part3.py`.

   Which of these classes will be the superclass? The subclasses?

2. You already identified the method(s) that are needed for the `NumberGame` class based on your reading of the starter code. Add these method(s) to your class bodies, with appropriate docstrings. A careful reading of the `NumberGame` class tells you not just what the method name should be, but also what the parameters of the method should be!

   If any of the methods should be abstract, make sure their body is `raise NotImplementedError`. Any abstract class should indicate that it is abstract in its class docstring.

### 3. Implement class `RandomPlayer`

Now that we have a `Player` class, we need one or more child classes that can complete its unimplemented method(s). Implement class `RandomPlayer` as a *subclass* of `Player`. Any `Player` methods that were not implemented must be overridden here in class `RandomPlayer`.

We have already imported module `random` for you.

### 4. Running the game!

Even though you only have one kind of player, you can still make the program run. In the `run_example_game` function, write code to:

1. Create a new `NumberGame` with a goal of 21, possible allowed moves between 1-3, and two random players.
2. Play the game.

Then call your `run_example_game` function in the Python console. It should be fun to watch two random players battle it out.

There will likely be small glitches to fix, but they will be things like forgetting an argument, and shouldn't be hard to fix. Read the error messages carefully—they include very precise information about what's wrong.

### 5: Add class `UserPlayer`

Now implement `UserPlayer`, which prompts the user for a move using the built-in function `input`. Note that this function returns a string, so you'll need to convert it to an integer by calling `int` on it.

Once you have `UserPlayer` done, modify the `run_example_game` function to play a game between one user player and one random player.

We hope you can beat the random player!

## Additional exercises

1. In your `ListMap` implementation from Part 2, try renaming these methods:

   - `size` into `__len__`
   - `assign` into `__setitem__`
   - `lookup` into `__getitem__`

   As you probably expect, these are *special methods* in Python, indicated by their double underscore names. If you rename these methods and have implemented them correctly, you should be able to execute the following code in the Python console:

   ```
   >>> my_map = ListMap()
   >>> my_map['a'] = 1    # Because of __setitem__
   >>> my_map['b'] = 2
   >>> my_map['c'] = 10
   >>> my_map['a']        # Because of __getitem__
   1
   >>> len(my_map)        # Because of __len__
   3
   ```

   And just like that, our `ListMap` instance looks a lot more similar to Python's built-in `dict` objects, at least as far as syntax is concerned. (But not efficiency—why?)

2. Try adding a third `StrategicAIPlayer` subclass to `Player`. This class should choose moves using a strategy that is *guaranteed* to win if this player goes first, and that can win if it goes second and its opponent does not play perfectly.

   To do this, you'll need to play a few games (stick with goal 21, moves 1-3), and feel free to discuss strategies with your classmates or your TA.

   If you have that "21-1-3" version of the game figured out, try generalizing the strategy to work for any goal, minimum and maximum. (How should you design the code if you can only offer a `StrategicPlayer` for the 21-1-3 version of the game?)

   Once you have `StrategicPlayer` implemented, update `make_player` one last time to give the user the choice of this third kind of player. Try running the game with a strategic and a random player. Does the strategic one always win?

3. Extend the `NumberGame` class to take more than two players.

4. Extend the `StrategicPlayer` class to find a winning strategy when there are more than two players. Is this even possible?! Read more about the classic game this part of the tutorial is based on here.