Chapter 3

Data and structures

This chapter dives deeply into some of Python's most popular data structures. We have previously studied lists, but just fixed-size lists. Python has robust means for creating, editing, and deleting elements from lists.

Dictionaries, on the other hand, are *associative lists* that allow complex data structures to be defined by associating data to objects, particularly strings. Finally, classes (used in *object-oriented programming*) are elegant data structures that associate data to functions. Object-oriented programming is beyond the scope of this course, but we will discuss some fundamental applications of these structures, which are found in practically all programming languages (such as structs in C).

It is important to note that all data types covered (except classes) in this chapter are called *iterable*. An iterable data structure is any structure that supports the *iter* protocols. This can be simply explained by the ability of using it in a `for` loop. The length of an iterable is computed by the function py `len(...)` .

1. Lists

Lists were already covered in the first section of this course. A list is a collection of items (usually with repetition) that are organized into contiguous locations and may be indexed using an integer. Python lists may store objects of various types and have dynamic sizes, allowing us to add and delete items from a list. Lists are mutable, that is, we can also change the items that are stored in a list.

Remark: Tuples are very similar to lists, but are immutable. In other words, once a tuple is created, it cannot be modified either by adding new elements or by modifying its elements.

1.1. Creation

There are many ways to create lists:

Using the list constructor

By calling the constructor py `list` , we can create a list from any iterable: another list, a tuple (or a dictionary as we will se later), etc. In the expression py `list(a)` such as `a` is a list or a tuple, the result is a copy of the argument. A call to py `list()` without argument returns an empty list. Alternatively, the function py `tuple` creates a tuple form any iterable data. As an example, the call py `list(range(10))` returns the list `[0,1,2,3,4,5,6,7,8,9]`.

Using the function `map`

This function has been introduced in the previous chapter. It applies a function to the elements of an iterable and returns a new list containing the results. For example, to produce a list of strings corresponding to the first 10 integers, we use `list(map(str,range(10)))`.

Using filters (the function `filter`)

Assume that `l` is a list of people's names, and we want to get a list of the names that start with `"a"` (small or capital). We can use the following code:

```python
def begins_with_a(l:list[str]) -> bool:
    res=[]
    for n in l:
        if n.lower().startswith('a'):
            res.append(n)
    return res
```

We didn't study the function `append` yet, so let's just say that this function adds an element to the end of a list. The function `begins_with_a` filters the elements from the list that satisfy the condition (`n.lower().startswith('a')`). In reality, this task is very common in programming, thus Python has a special function that that makes it easier to create filtered content. Using the `filter` function, we can write the following code: `list(filter(lambda n:n.lower().startswith('a'),l))`.

Using comprehensions

Although mapping with `map` and filtering with `filter` can handle out most of the business about lists, this is not a *pythonic* way to do things. Instead, we prefer using comprehensions, which is an elegant way to map, filter or do anything we want with lists. A comprehension is a way to perform some processing on the elements of a list (or any iterable in fact). The general form of a comprehension is `[processing for element in iterable]`.

For instance, to generate a list of string representations of integers up to 10 without the function `map`, we would write `[str(i) for i in range(10)]`. Comprehensions are so powerful that we can easily nest them all together. Consider, for example, producing a string representation of a matrix:

```python
def stringify_matrix(mat):
    return "\n".join([",".join([str(i) for i in row]) for row in mat])
```

It may appear odd at first, but after we get used to it, comprehensions will be a useful tool for any Python coder. Comprehensions can, hence, replace, the function `map`.

A comprehension can be used to filter the items of a list by using this syntax `[processing for element in iterable if condition]`, where `condition` is a boolean expression. For example, the function that filters the names starting with "a" is rewritten as follows (we won't use the `append` function):

```python
def begins_with_a(l:list[str]) -> bool:
    return [n for n in l if n.lower().startswith('a')]
```

Using the function `zip`

In reality, the function `zip` is not a constructor of lists but combines them. Given two lists `a` and `b`, `list(zip(a,b))` produces a list of tuples by combining the elements of `a` and those of `b`. For example, `list(zip([0,1,2],[3,4,5]))` yields the list `[(0,3),(1,4),(2,5)]`. This makes it possible to write a simple function to compute the scalar product of two vectors:

```python
def scalar_product(a:list[float],b:list[float]) -> float:
    return sum([u*v for u,v in zip(a,b)])
```

Remark: When iterating over table with the loop `for`, indices are not available by default. To get the indices together with the element, it is possible to use the following code:

```python
for i,n in zip(l,range(len(l))):
    # do some stuff
```

However, an easier way to do that is to use the function `enumerate`:

```python
for i,n in enumerate(l):
    # do some stuff
```

1.2. Manipulation and update

List items can be manipulated by specifying their indices. For example, `l[5]=0` assigns the value 0 to the sixth element of the list `l`.

Never try to access items beyond the last one of a list (use the `len` function to determine the length of a list).

Negative indices are supported and are used to access the items in reverse order, i.e., the final element is at index −1, the penultimate element is at index −2, etc.

Slicing is another way for accessing (and extracting parts of a list). A slice is a part of a list that allows for extracting a portion of the list from a given position ‘a’ (included) to a given position ‘b’ (excluded) using the syntax `l[a:b]`. It is possible to omit the first position (in this case, it will be 0) and/or the last end position (in this case, it will be the length of the list). Hence, a very simple code to produce a copy of a list is `l[:]`. The slicing supports a step argument. For example, py `l[::2]` returns a list with the elements of the list `l` at even positions. When used with negative indices, it is possible to construct interesting things, such as the reverse of a list using py `l[::-1]`. Finally, slices can be used to insert a list at a certain position in the list, but we prefer to use the `insert` method (at least for inserting one element).

Insertion

A list can be extended by adding new elements. There are mainly three functions to do that:

1. The `append` function: py `l.append(x)` adds the element `x` at the end of the list `l`.
2. The `insert` function: py `l.insert(i,x)` adds the element `x` at the position `i` in the list `l`. The elements at positions `i` and higher are shifted to the right.
3. The `extend` function: py `l.extend(lp)` adds all the elements of the list `lp` at the end of the list `l`.

To illustrate these functions, assume a list of integers `l=[1,2,3]` and a list of integers `lp=[4,5,6]`:

1. py `l.append(4)` gives `l=[1,2,3,4]`.
2. py `l.insert(1,4)` gives `l=[1,4,2,3]`.
3. py `l.extend(lp)` gives `l=[1,4,2,3,4,5,6]`.

1.3. Deletion

Many options are offered to delete an element from a list.

Using the function `remove`

The function `l.remove(x)` removes the first occurrence of the element `x` in the list `l`. If the element is not found, it raises an exception. For instance, `[1,4,3].remove(4)` yields the list `[1,3]`.

Using comprehension

Comprehensions can be used to delete all occurrence of an element `x` from a list `l` by using the syntax `[y for y in l if y!=x]`. Note that we are creating a copy of the list and not modifying it directly.

Using the function `pop`

The function `pop(i=-1)` removes an element from the position `i` and returns it (an exception is raised if the index is out of range). All elements from position `i+1` to the end of the list are shifted to the left. When called without arguments, this function removes the last element of the list.

Using the function `del`

The statement `del l[i]` removes the element at position `i` from the list `l`. Be careful with this statement since `del a` will permanently remove the variable `a` from the namespace.

Using the function `clear`

The function `l.clear()` removes all the elements from the list `l`.

Using slices

Slices can be used to remove elements from a list. Let's write an equivalent function to `pop` (the function does not work for `i=-1`):

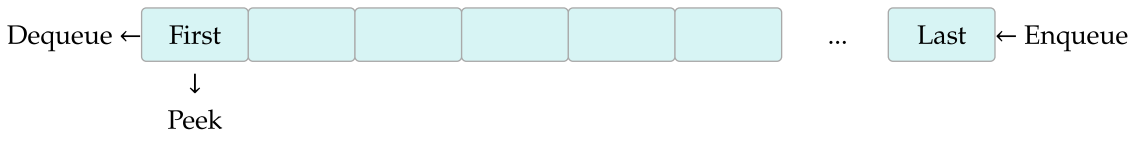
```python
def my_pop(l,i):
    return l[:i]+l[i+1:]
```

1.4. Applications: queues and stacks

Lists are widely used data structures that find application in many areas. One such application involves queues and stacks. These structures enable the storage of produced data in specific manners, after which the data is consumed in a different way. Let's examine these structures and how they can be implemented using Python lists.

Queues

A queue is a *First In First Out* data structure; that is, the first element added to the queue is the first one to be removed. Such a structure has four main operations: *enqueue* (add a new element at the end), *dequeue* (remove the first element, and all elements will be shifted to the left), *peek* (return the first element without removing it), and *is_empty* (return `True` if the queue is empty).



The four functions can be implemented as follows (for a queue of integers):

```python
def enqueue(l:list[int],x) -> list[int]:
  l.append(x)
  return l
def dequeue(l:list[int]) -> int:
  return l.pop(0)
def peek(l:list[int]) -> int:
  return l[0]
def is_empty(l:list[int]) -> bool:
  return len(l)==0
```

Example 3.1 : History of purchases

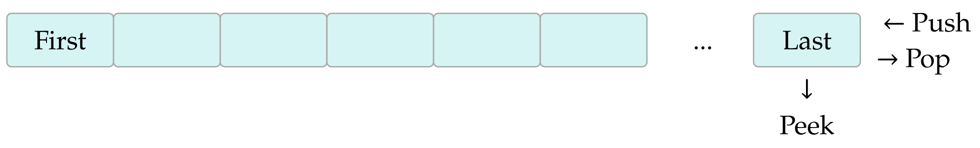
Let's look at queues as an example. Consider a program that reads a shop's daily income and saves it in a queue just to be printed later. Each purchase is described as a series of product prices purchased by the customer, followed by -1 to indicate the end of the transaction. . The end of operations is made by reading the string `quit`.

```python
def read_write_purchases():
    def print_current_purchase(purchases):
        while not is_empty(purchases):
            print(dequeue(purchases))

    purchases = []
    price=""
    while price!="quit":
        price = input("The price of the next product (-1 to end a purchase, quit to end)? ")
        if price!="quit":
            price=float(price)
            if price==-1:
                print_current_purchase(purchases)
            else:
                enqueue(purchases,price)
    print_current_purchase(purchases)
```

Stacks

A stack is a First In Last Out data structure; that is, the last element added to the queue is the first one to be removed. Such a structure has four main operations: *push* (add a new element at the end), *pop* (remove the last element), *peek* (return the last element without removing it), and *is_empty* (return `True` if the queue is empty).



The four functions can be implemented as follows (for a stack of integers):

```python
def push(l:list[int],x) -> list[int]:
  l.append(x)
  return l
def pop(l:list[int]) -> int:
  return l.pop()
def peek(l:list[int]) -> int:
  return l[-1]
def is_empty(l:list[int]) -> bool:
  return len(l)==0
```

Example 3.2 : Reverse a list using stacks

Thanks to the *First In Last Out* strategy, it is possible to reverse a list by using a stack. The idea is to push all the elements of the list into a stack and then pop them out one by one.

```python
def reverse(l:list[int]) -> list[int]:
    s = []
    for e in l:
      push(s,e)
    res=[]
    while not is_empty(s):
      res.append(pop(s))
    return res
```

Remark: Python has a list-like type that avoids repetition. It is called `set` and is built with the braces (`{}`) instead of the brackets. Sets are generally used to store unique values; they support set operations like union, intersection, difference, symmetric difference, etc. Sets are iterable but are not subscriptable; that is, we can not access the elements of a set s by using the classic syntax `s[i]`. A set element can not be changed although the set is mutable (we can add or drop elements).

2. Dictionaries

Lists are great structures that can store several types of data. However, indexing data using integers is not always practical. For instance, consider a student who is characterized by his name, age, and address. To represent a student called Ali, aged 20 years and living in Annaba, one can use py `'st=[Ali',20,'Annaba']`. Notice that, despite this being possible, it is not very readable, since the name of the student would be st[0]!

Python has another type that allows indexing data with keys. A key can be any immutable type: integers, floats, strings, tuples, etc. However, most of the time, we use strings. Values that can be stored within a dictionary can be of any type, including lists, dictionaries, etc. This allows for modeling complex situations involving compound data structures.

Dictionaries are mutable and are iterable. We can add, update, or delete elements from a dictionary. Likewise, we can iterate over the keys, the values, and/or the keys/values of a dictionary.

2.1. Creation

Dictionaries can basically be created using braces (as for sets). For instance, the student is created as follows: py `st={'name':'Ali','age':20,'address':'Annaba'}` . Now, to know the name of the student, we use the syntax `st['name']`, which is by far more concise than lists. In this case, `'name'` is the key and `'Ali'` is the value.

Another way to create dictionaries is to use the function `dict`. This function takes an arbitrary number of arguments with arbitrary names. For instance, `dict(name='Ali',age=20,address='Annaba')` is equivalent to `{'name':'Ali','age':20,'address':'Annaba'}`. But how is that possible? Let's simulate the function `dict`:

```python
def my_dict(**kwargs) -> dict:
  return kwargs
```

The function `my_dict` is very simple; it just returns its arguments. ** (used to unpack arguments) means that the function takes an arbitrary number of arguments with arbitrary names, but the function sees the whole as one dictionary. The unpacking of dictionary makes it possible to easily create a copy of a dictionary just by using `{**d}`.

Mapping, filtering, and slicing are not available for dictionaries. But it is possible to map or filter using the two functions `keys()` and `values()` which return, respectively, the keys and the values of a dictionary. For instance, to filter the values of a dictionary, we can write:

```python
def filter_values(d:dict,f:(Any)->bool) -> dict[str,int]:
    res={}
    for k in d.keys():# this is equivalent to do for k in d
        if f(d[k]):
            res[k]=d[k]
    return res
```

Again, comprehensions are the pythonic way to make all kinds of manipulations over iterables. For instance, we can filter the values of a dictionary as follows: `{k:d[k] for k in d if f(d[k])}`. Thanks to comprehension, it is possible to make a copy of a dictionary with `{k:d[k] for k in d}`.

2.2. Iteration

As explained in the previous section, there are many ways to iterate over dictionaries. For instance, we can iterate over the keys by using two equivalent syntaxes: `for k in d.keys()` and `for k in d`. We can also iterate over the values by using `for v in d.values()`. To iterate over both keys and values, we use `for k,v in d.items()`. Another ugly way to accomplish that is `for k,v in zip(d.keys(),d.values())`.

2.3. Manipulation and update

To access or modify an entry of a dictionary, we just have to use `d[key]`. For example, let `products` be a dictionary of products associated with their prices `products={"milk":130,"rice":"100","flour":90}`. To change the price of milk to 125, we just have to write `products["milk"]=125`. Another possibility to update an entry of a dictionary is to use `{**products,**{"milk":125}}` (note, however, that we are making a new copy of the original dictionary here).

However, if the key does not exist, a `KeyError` exception is raised. As a result, we should always check for the existence of a key in a dictionary by using the expression `key in d` before trying to read the value associated with `key`. Another method consists of using the function `get` which returns `None` if the key is not found. For instance, `products.get("milk")` returns 130 and `products.get("eggs")` returns `None`.

It is possible to update many entries of a dictionary at once by using the function `update`. For instance, `d.update({'milk':125,'eggs':100})` will update the price of milk to 125 and the price of eggs to 100.

2.4. Deletion

A dictionary stores all sorts of values, including `None`. So, associating this value with a key does not remove the key from the dictionary. Instead, we can use the `del` statement to delete an entry of a dictionary. For instance, `del d['milk']` will delete the entry `milk` from the dictionary `d`. It is also possible to create a new dictionary without an entry by using `{k,v for k,v in d.items() if k!='milk'}`.

The function `pop` can also be used to delete an entry of a dictionary. For instance, `d.pop('milk')` will delete the entry `milk` from the dictionary `d` and return its value. The function `clear` deletes all the entries from a dictionary.

Example 3.3 : List of students

As an example, let's consider a list of students. We want to write three functions:
1. A function that looks for a student by name (if the student does not exist, then return `None`)
2. A function that sorts the students according to a given attribute
3. A function that returns a list of students whose names start with a given letter

The code of the three functions may be the following:

```python
def find_student(students:list[dict],name:str) -> dict:
  for s in students:
    if s['name']==name:
      return s
  return None

def sort_students(students:list[dict],attr:str) -> None:
  students.sort(key=lambda s:s[attr])

def filter_students(students:list[dict],letter:str) -> list[dict]:
  return [s for s in students if s['name'][0].lower().startswith(letter)]
```

3. Classes

Python is an object-oriented language, which means that each value in Python is an object. So what is an object? An object is a combination of data and functions. For instance, the value 1 in Python is associated with the integer value 1 and a set of functions: `__abs__`, `bit_length`, etc. This is a way to know what functions that can be applied to a given type.

A class is a template of an object; i.e., it is the declaration of the structure of an object and its functions (called methods). Studying object-oriented programming is beyond the scope of this course, but we will study a special case of them, called dataclasses, since they are very similar to dictionaries in Python and structures in C.

3.1. Dataclasses

A dataclass is a special class that consists mostly of data (with too few methods). Let's use the student example to demonstrate how dataclasses are built and implemented. A student is identified by his name (a string), age (an integer), and address (a string). To create a dataclass that corresponds to that, we use the code below:

```python
#first we have to import dataclass since it is not a basic type
from dataclasses import dataclass

class Student(dataclass):
  name: str
  age: int
  address: str
```

This declaration allows us to construct an instance of the class Student as follows: py `st=Student('Ali',22,'Annaba')` (this is known as the object's *constructor*). To get the student's name, we use the dot notation: st.name (compare this to the dictionary syntax `st["name"]`). In this code, `name`, `age` and `address` are the *attributes* of the object `st`.

The constructor is very flexible since it also support dictionary assignment via py `st=Student(name='Ali',age=22,address='Annaba')`. This allows for creating objects from dictionaries just by using the unpacking syntax.

Dataclasses are more compact than dictionaries since we won't use the string quotes. However, we cannot use any attribute name (as this is the case for dictionaries). Only valid Python names can be used. However, we can specify default values. For instance, by using `address:str="Annaba"`, we specify that if the instantiation does not provide an address, then its value will be `"Annaba"`.

Another difference between dictionaries and dataclasses is iterability. While it is possible to iterate over a dictionary, dataclasses are not iterable by default. It is possible to make an iterable class, but this won't be covered by this course.

Dataclasses provides a special function called `__str__` allowing to generate a simple string representation of an object. This function is systematically called whenever we want to transform an object into a string. For example, py `print(st)` will produce `Student(name='Ali', age=22, address='Annaba')` (pretty similar to a dictionary). It is also possible to transform a dataclass into a dictionary by using the function `asdict`. Hence, py `print(asdict(str))` (we have to import `asdict` first) will produce the following output: `{'name': 'Ali', 'age': 22, 'address': 'Annaba'}`.