5. Collections

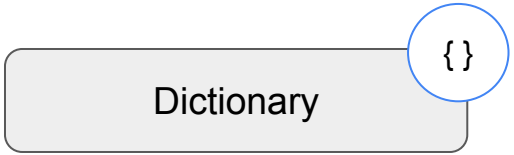
More Data Types

What are Collections?

Remember our primitive data types? There are more complex ones!

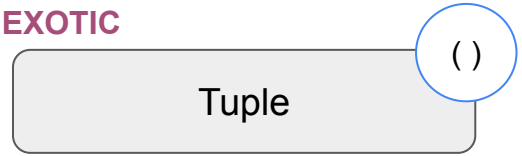
`[1, 2, 5, 0, 4712]` or `["a", 5, "cde"]`

`{1, 2, 5, 0, 4712}` or `{"a", 5, "cde"}`



`{"x":1, "y":2, "z":3}`

EXOTIC



`(1,8)` or `("a", "b")` or `(0.5, "x")`

Lists

- **ordered**: the list holds the elements in the order you add them

```
mylist = [2, 5, 3]
mylist.append(19)
print(mylist)
>>> [2, 5, 3, 19]
```

- **mutable**: you can modify the list
- **allows duplicates**:

```
[2, 5, 2]          is valid
["x", "x", 3]      is valid
```

Working with Lists

Adding elements to a list: **`.append()`**

```
mylist = [“Alice”, “Bob”]
mylist.append(“Charlie”)
print(mylist)
>>> [“Alice”, “Bob”, “Charlie”]
```

Removing elements from a list: **`.remove()`**

```
mylist.remove(“Bob”)
print(mylist)
>>> [“Alice”, “Charlie”]
```

Working with Lists: Indexing

Indexing

```
mylist = [“Alice”, “Bob”, “Charlie”]
```

index = 0 1 2

We can use an **element’s index** to **refer to it**!

```
print(mylist[0])
```

```
>>> “Alice”
```

Working with Lists: Indexing

```
mylist = [“Alice”, “Bob”, “Charlie”]
```

index	=	0	1	2
reverse index	=	-3	-2	-1

```
print(mylist[-1])
```

```
>>> “Charlie”
```

Working with Lists: Using Indices

Now that we can specify our elements, we have more options to work with!

```
mylist = [“Alice”, “Bob”, “Chalie”]
```

We made a typo with “Chalie”. How can we fix it?

```
mylist[-1] = “Charlie”
```
or
```
mylist[2] = “Charlie”

print(mylist)
>>> [“Alice”, “Bob”, “Charlie”]
```

Working with Lists: Using Indices

We can also delete elements based on their index.

```
mylist = [“Alice”, “Bob”, “Charlie”]

mylist.pop(1)

print(mylist)


 >>> [“Alice”, “Charlie”]
```

Working with Lists: Index Ranges

Indices can also specify a range of elements.
General syntax: `[start:end]`

```
mylist = [“apple”, “cookie”, “tea”, “coffee”]
```

We want to get the first two elements of the list. How?

```
print(mylist[0:2])
>>> [“apple”, “cookie”]
```

NOTE

The end index is excluded!
`[0:2]` -> returns elements at index `0` and `1`, but `2` is excluded.

Working with Lists: Index Ranges

Let’s make it a little more difficult…

```
mylist = [“apple”, “cookie”, “tea”, “coffee”]


print(mylist[1:])
 >>> [“cookie”, “tea”, “coffee”]


 print(mylist[:1])
  >>> [“apple”]
```

Working with Lists: Index Ranges

```
mylist = [“apple”, “cookie”, “tea”, “coffee”]

print(mylist[-2:])
 >>> [“tea”, “coffee”]


 print(mylist[:])
  >>> [“apple”, “cookie”, “tea”, “coffee”]
```

Exercise: Lists

1. Define a list called `alphabet`. It should contain the first 5 letters of the alphabet.
2. Define the variables `first_letter` and `last_letter`. Fill them accordingly.
3. Define the variable `first_three`. Use slicing to fill this variable.
4. Define the variable `last_three`. Use slicing to fill this variable.
5. Add the next 3 letters of the alphabet to our `alphabet`.
6. Now, update your variable `last_three`.
7. Delete the last letter from our `alphabet`.

Print out your variables whenever you need to!
You'll see what your code is doing.

Exercise: Lists, Solution

```
alphabet = [“a”, “b”, “c”, “d”, “e”]

first_letter = alphabet[0]
last_letter = alphabet[-1]

first_three = alphabet[:3]          alternative: alphabet[0:3]
last_three = alphabet[-3:]

alphabet.append(“f”)
alphabet.append(“g”)
alphabet.append(“h”)

last_three = alphabet[-3:]    >>> [“f”, “g”, “h”]

alphabet.remove(“h”)
```

Sets

- **unordered**: there's no specific order to the elements

```
myset = {1, 7, 5}
print(myset)
>>> {1, 5, 7}
```

- **mutable**: you can modify the set
- **doesn't allow duplicates**:

```
myset = {1,1}
print(myset)
>>> {1}
```

Working with Sets

Most important feature: **no duplicates!**

Say, we have a bunch of numbers, and we want to save them in a data structure.

```
1, 2, 1, 3, 2, 1, 3
```

But we want only unique numbers. What's the better choice, and what's the result of each of the following:

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

myset =   {1, 2, 1, 3, 2, 1, 3}      >>> {1, 2, 3}
```

Working with Sets

We can also make sets like this:

```
myset = set(...)
```

For example, to **convert** an **existing** list to a set:

```
list_with_duplicates = [1, 2, 1, 3]

no_duplicates = set(list_with_duplicates)

print(no_duplicates)


 >>> {1, 2, 3}
```

NOTE

Converting an object from one type to another is called ***casting***. It also works for many other types.
E.g. `float("123") >>> 123.0`

Working with Sets

Remember indexing from our lists? Try out if this works with sets, too.

- Declare a set
- Try printing the first element in the set
- Tell me what happens

```
>>> TypeError: 'set' object is not subscriptable
```

Why do you think that might be the case? Why can't we index sets?

→ *Remember: sets are **unordered**!*

Dictionaries

You can **map** values to other values. For example, let's say we want to represent people's ages. We could do:

```
aliceAge = 25

bobAge = 30
```

With a Dictionary, we can collect this information in one place:

```
allAges = {"Alice": 25, "Bob": 30}
```

key value key value

Dictionaries are unordered and mutable. However, **the keys need to be unique**!

Dictionaries… why is this cool again?

We don’t just save data in some data structure.
We even save **relations** between this data!

```
allAges = {“Alice”: 25, “Bob”: 30}
```

If we want to access the age of Alice now, we can:

```
alicesAge = allAges[“Alice”]

print(alicesAge)

>>> 25
```

Dictionaries… why is this cool again?

```
allAges = {“Alice”: 25, “Bob”: 30}
```

Try printing the age of Bob!

```
bobsAge = allAges[“Bob”]
print(bobsAge)
>>> 30
```

By the way… we don’t need to declare `bobsAge` necessarily:

```
print(allAges[“Bob”])
>>> 30
```

Dictionaries: keys and values

We can access dictionaries through their keys. We can probably do the same with values:

```
allAges = {“Alice”: 25, “Bob”: 30}

print(allAges[25])
```

- Tell me what you think will happen
- Actually try it out

```
>>> KeyError: 25
```

Getting information from a dictionary doesn’t work both ways!

key -> value 

value -> key 

Exercise: Dictionaries

We have a menu at a restaurant.

The available dishes are: pasta, pizza, salad, wine, water
Their prices are: 10.50, 9.00, 6.50, 4.00, 2.30

1. Save this information into a dictionary called `menu`.
2. From `menu`, access the prices of pasta and wine. Save them in separate variables.
3. Print the prices of pasta and wine.

Exercise: Dictionaries, Solution

The available dishes are: pasta, pizza, salad, wine, water
Their prices are: 10.50, 9.00, 6.50, 4.00, 2.30

```
menu = {“pasta”: 10.50, “pizza”: 9.00, “salad”: 6.50,
        “wine”: 4.00, “water”: 2.30}

pasta_price = menu[“pasta”]
wine_price = menu[“wine”]

print(pasta_price)
print(wine_price)
```

Dictionaries: adding elements

We forgot dessert! Let’s add cake to our menu.

```
menu = {“pasta”: 10.50, “pizza”: 9.00, “salad”: 6.50,
        “wine”: 4.00, “water”: 2.30}
```

We can do it very straightforward:

```
menu[“cake”] = 3.50
```

Try it, and print your menu again!

Dictionaries: removing elements

To remove an entry, we can use `.pop()`

```
menu = {“pasta”: 10.50, “pizza”: 9.00, “salad”: 6.50,
        “wine”: 4.00, “water”: 2.30}
```

Try removing salad from our `menu`.

```
menu.pop(“salad”)
print(menu)
```

```
>>> {“pasta”: 10.50, “pizza”: 9.00, “wine”: 4.00,
     “water”: 2.30, “cake”: 3.50}
```

Dictionaries: modifying existing entries

Besides adding/deleting, we can change existing entries.

```
menu = {“pasta”: 10.50, “pizza”: 9.00, “wine”: 4.00,
        “water”: 2.30, “cake”: 3.50}
```

We’re feeling particularly nice, and want to lower the pizza price to 8.00.

Any idea how we might do that?

```
menu[“pizza”] = 8.00

print(menu)
>>> {“pasta”: 10.50, “pizza”: 8.00, “wine”: 4.00,
     “water”: 2.30, “cake”: 3.50}
```

Recap

- Lists `[1, 2, 1, 3, "a"]`
- Sets `{1, 2, 3, "a"}`
- Dictionaries `{"a": 1, "b": 2}`

Indexing:

```
mylist[0]                  -> first element
mylist[-1]                 -> last element
mydictionary["key"]        -> value associated with this key
```

Modifying:

```
mylist.append("new")       -> adds new element
mylist.remove("new")       -> removes element

myset.add("new")           -> adds new element
myset.remove("new")        -> removes element

mydictionary["new"] = value
                           -> adds new element
mydictionary.pop("new")
                           -> removes element
```