Part 5

Introduction to Programming

3.10.2024

Last week

- parameters & arguments
- return values of functions
- type hints
- lists
- iteration with for statement
- range function
- count and replace
- f strings

About the Exercises on This Course

Becoming a proficient programmer requires a lot of practice, sometimes even quite mechanical practice. It also involves developing problem solving skills and applying intuition. This is why there are a lot of exercises of different kinds on this course. Some of them ask you to quite straightforwardly apply what you have learnt in the material, but some of them are intentionally more challenging and open-ended.

Some of the exercises might at first seem overwhelming, but this is nothing to worry about. None of the exercises is strictly mandatory, and in fact *only 25 % of the points in each part is required to pass the course.* You can find more details about passing the course on the page on grading.

The exercises are not in any specific order of difficulty. Each section usually introduces some new programming concepts, and these are then practised with both simpler and more complicated exercises. **If you come across an exercise that feels too difficult, move on to the next one.** You can always come back to the more difficult exercises if you have time later.

When the going inevitably gets tough, a word of consolation: a task that seems impossibly difficult this week will likely feel rather easy in about four weeks' time.

Items other than integers

Lists can hold any items, for example strings

```
names = ["Marlyn", "Ruth", "Paul"]
print(names)
names.append("David")
print(names)

print("Number of names on the list:", len(names))
print("Names in alphabetical order:")
names.sort()
for name in names:
  print(name)
```

Items other than integers (2)

...or floats

```
measurements = [-2.5, 1.1, 7.5, 14.6, 21.0, 19.2]

for measure in measurements:
    print(measure)

mean = sum(measurements) / len(measurements)

print("The mean is:", mean)
```

Nested lists

List items can be lists:

```
my_list = [[5, 2, 3], [4, 1], [2, 2, 5, 1]]
print(my_list)
print(my_list[1])
print(my_list[1][0])
```

Matrices

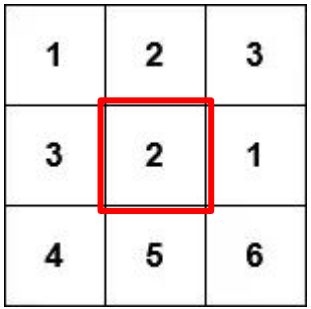
Nested lists are a good way to model matrices:

1	2	3
3	2	1
4	5	6

```
my_matrix = [[1, 2, 3], [3, 2, 1], [4, 5, 6]]
```

Referencing items

Referencing an item in the matrix now requires two indices

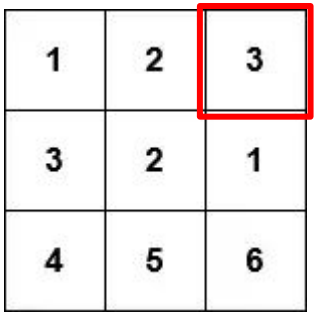


First if for **the row**, the second is for the **item in the row**

```
print(m[1][1])
```

Referencing items

Referencing an item in the matrix now requires two indices

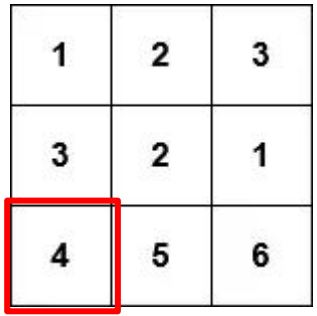


First if for **the row**, the second is for the **item in the row**

```
print(m[0][2])
```

Referencing items

Referencing an item in the matrix now requires two indices



First if for **the row**, the second is for the **item in the row**

```
print(m[2][0])
```

Iterating matrix

Iterating matrix can be done with two nested for loops

First is for iterating rows...

...and the second for iterating items in rows

```
my_matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for row in my_matrix:
    print("a new row")
    for element in row:
        print(element)
```

Changing item values

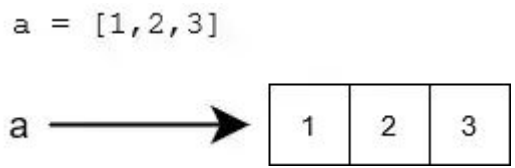
If we need to change item values, we can use two nested for loops that utilize the range function

```
m = [[1,2,3], [4,5,6], [7,8,9]]

for i in range(len(m)): # using the number of rows in the matrix
    for j in range(len(m[i])): # using the number of items on each row
        m[i][j] += 1

print(m)
```

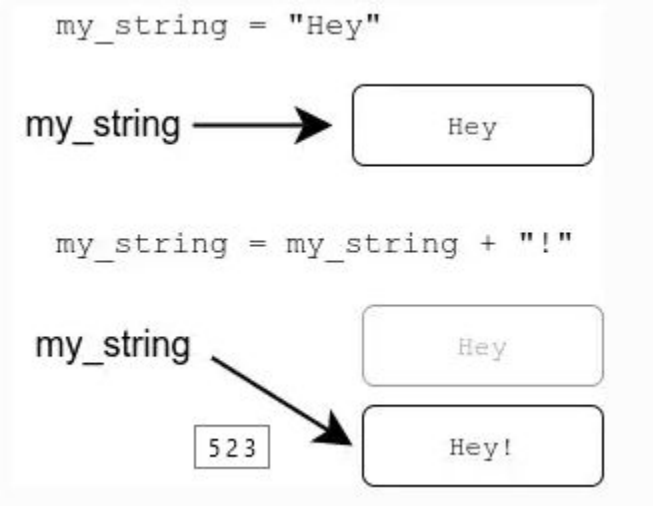
References

In Python, all variables are references

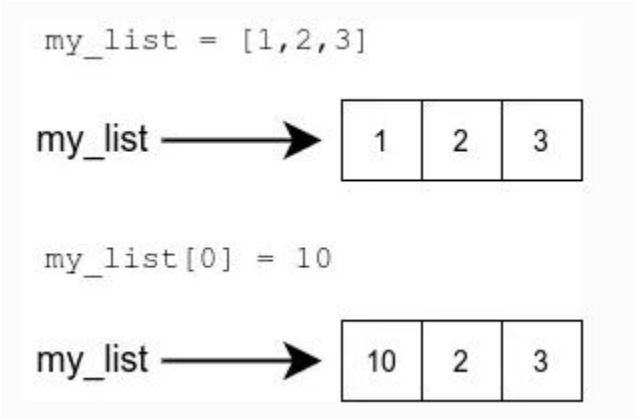
The **value** of the variable is a **reference** to an **object**



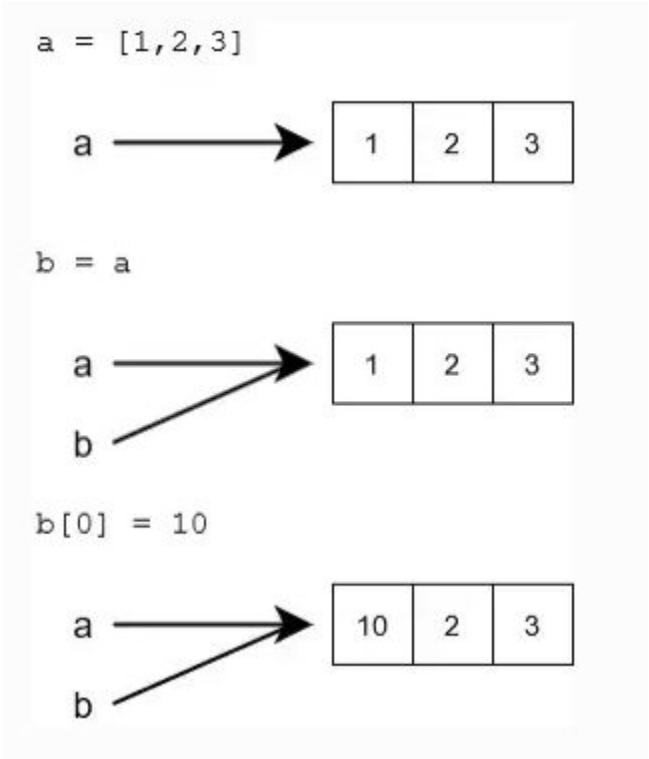
Strings are immutable



Lists are mutable



Lists are mutable (2)



Copying a list

List can be copied by taking a slice (a "sublist") which contains all items

...e.g. `my_list[0 : len(my_list)]`

...or just `my_list[:]`

List as a function argument

If the argument is a list reference, the function can change the list

```
def add_item(my_list: list):
    new_item = 10
    my_list.append(new_item)

a_list = [1,2,3]
print(a_list)
add_item(a_list)
print(a_list)
```

Side effects

If function changes a list (and this is not the main purpose of the function), this is called a **side effect**

```
def second_smallest(my_list: list) -> int:
    # in an ordered list, the second smallest item is at index 1
    my_list.sort()
    return my_list[1]

numbers = [1, 4, 2, 5, 3, 6, 4, 7]
print(second_smallest(numbers))
print(numbers)
```

Dictionary

In dictionary, all items consist of key-value pairs

Dictionary is useful (and fast) when we want to access values based on the keys

Setting items

Dictionary is initialized with curly brackets

Setting an item adds or changes the value

Items are referenced like in a list, but instead of index we use keys

```
results = {}
results["Mary"] = 4
results["Alice"] = 5
results["Kim"] = 2
```

About items

Key must be **immutable**

Hence a list for example cannot be used as a key

Iterating a dictionary

Dictionary can be iterated with a for statement

For statement iterates through the keys in the dictionary, one at a time

```
translations = {}

translations["apina"] = "monkey"
translations["banaani"] = "banana"
translations["cembalo"] =
"harpsichord"

for key in translations:
    print("key:", key)
    print("value:", translations[key])
```

Method items

If we want to iterate through both, keys and values, we can use the method **items**

```
for key, value in
translations.items():
    print("key:", key)
    print("value:", value)
```

Removing items

With **del** statement...

```
personnel = {"Antti": "principal", "Emilia": "professor", "Arto": "lecturer"}
del personnel["Arto"]
print(personnel)
```

or with a method **pop**

```
personnel = {"Antti": "principal", "Emilia": "professor", "Arto": "lecturer"}
removed = personnel.pop("Arto")
print(personnel)
print("Removed:", removed)
```

Dictionary in data grouping

Single dictionary can be used to group information that belongs together

```
person= {"name": "Paula Python", "length": 154, "weight": 61, "age:" 44}
```

Tuple

A list-like data structure

Syntactical differences:

- tuple is notated with parentheses (), list with brackets []
- tuple is immutable

Why tuple?

Tuple is a collection of values that are connected

List is a collection of similar items, and the size (or order) may change

Next week

Reading and writing files

Errors

Local and global variables