Python from Zero

More Data Types

Recap

Fundamental data types

Type	Description
`int`	"Whole numbers", e.g. 0, 1, 2, 3, -1, -2, -3, ...
`float`	Floating point numbers, e.g. 3.14, 0.1, -1.23
`bool`	Boolean, can only have the values `True` or `False`
`string`	Holds text values (actually not that basic)

```
my_int = 1
my_float = 3.14
my_bool = True
my_str = "Hello World!"
```

Recap

Basic operations

```
1 + 1 # adding numbers
2 - 1 # subtraction
3 * 5 # multiplication
3 / 4 # division, note that `x / 0` for any x is illegal

"Hello " + "World" + "!" # string concatenation
```

Fun with booleans

```
bool(0)   # 0 and empty string "" evaluate to False, otherwise True
int(True) # True is converted to 1, False to 0
```

Recap

Functions

```
def f(x):
    result = 2*x + 5
    return result

f(3)
```

```
11
```

More Datatypes

So far, we know `int`, `float`, `bool` and `str`

Variables of these types can only hold a single value

There are more datatypes in Python that can hold *multiple values*!

- `tuple` - immutable collection of values
- `list` - mutable, dynamic list of values
- `set` - mutable, dynamic set of values
- `dict` - mutable, dynamic collection of key-value pairs

Tuples

Immutable, ordered list of elements, i.e.

- Fixed size, cannot be changed
- Elements cannot be changed
- The elements remain at fixed positions in the list

However, element types may differ

```
x = (1,2,3)                        # x is a tuple
y = tuple(("a", "b", "c", 6)) # y is also a tuple (note the double parantheses)
```

Tuples

Element Access

Use the `[ ]` operator to access the individual elements of a tuple!

- Elements in a tuple are counted, **starting from zero**
  - `[0]` thus accesses the **first** element, `[1]` the second, and so on
- You can access elements from behind with negative numbers
  - `[-1]` accesses the **last** element, `[-2]` the second last, etc.
- These numbers are called an **index**

Tuples

Element Access

Question: what is the highest *index* in this tuple?

```
(7, "orange", 3.141, "bread", "leite", 42)
```

Tuples

Element Access

Question: what is the highest *index* in this tuple?

```
  (7, "orange", 3.141, "bread", "leite", 42)
#  0  1         2      3        4        5
```

Answer: 5, because there are six elements in the tuple and indexing starts from zero

Tuples

Element Access

```python
x = ("a", "b", "c", 6)
print(x[0])
print(x[1])
print(x[2])
print(x[3])
print(x[4]) # make sure you don't do this!
```

```
a
b
c
6
IndexError: tuple index out of range
```

Again, code along if you like!

Tuples

Element Access

```
x = ("a", "b", "c", 6)
print(x[-1])
print(x[-2])
print(x[-3])
print(x[-4])
print(x[-5]) # make sure you don't do this!
```

```
6
c
b
a
IndexError: tuple index out of range
```

Tuples

A useful function for tuples (and lists, sets, dicts): `len()`

- Return the number of elements inside a tuple

```
len(())                                            # 0
len((1,2,3))                                       # 3
len((7, "orange", 3.141, "bread", "leite", 42))    # 6
```

The highest index of a tuple `x` *is always* `len(x)-1` (unless `x` is empty)

Slicing

It is possible to access not just a single element, but a *range* of elements from a tuple!

Say `var` is a tuple and `a` and `b` are valid indices, then

- `var[a:b]` - yield elements with indices `a, a+1, a+2, ..., b-1`
- `var[a:b:n]` - yield elements `a, a+n, a+2n, ...` up to, but not including, `b`

If you omit `a`, slicing starts with the first element
If you omit `b`, slicing continues until including the last element

- `var[::-1]` - yield elements in reversed order

Slicing

```
x = (7, "orange", 3.141, "bread", "leite", 42)
print(x[1:3])  # elements at index 1, 2, but NOT 3!
print(x[1:])   # all elements from index 1 on
print(x[:3])   # all elements up to, but NOT INCLUDING 3
print(x[-2:])  # all elements from the second last index on
print(x[::-1]) # all elements in reverse order
print(x[::2])  # every second element in normal order
print(x[1::2])  # every second element in normal order, starting with the second element
```

```
('orange', 3.141)
('orange', 3.141, 'bread', 'leite', 42)
(7, 'orange', 3.141)
('leite', 42)
(42, 'leite', 'bread', 3.141, 'orange', 7)
(7, 3.141, 'leite')
('orange', 'bread', 42)
```

Nested Tuples

For a tuple, it does not matter what *type* the elements are

- You can store a tuple inside a tuple!

```
x = (1, 2, ("a", "b", "c"))
print(x[0])
print(x[2])
```

```
1
('a', 'b', 'c')
```

Nested Tuples

To access the elements in the inner tuple, just use `[ ]` again!

```
x = (1, 2, ("a", "b", "c"))
print(x[2][0])
```

```
'a'
```

You can nest as many tuples as you ~~like~~ have memory available

```
x = (1, 2, ("a", (True, False, (3.14, "got me")), "c"))
print(x[2][1][2][1])
```

```
got me
```

It's Your Turn!

- Please continue reading the notebook, read the "Tuples" section

Do the exercise!

Exercise:
Define a shopping list as nested tuples. Group vegetables, fruits and diary products in seperate tuples. Then print messages, indicating if "Fennel" is in one of the tuples and count the total number of items on the shopping list.

Hint: use `in` to check if something is contained in a tuple:

```
"foo" in ("foo", "bar") # returns True
"baz" in ("foo", "bar") # returns False
```

Lists

Like tuples:

- List of multiple values
- Values remain at a fixed position
- Type of values does not matter and can be mixed

But different:

- Are *mutable and dynamic*, i.e. you can
    - Append elements
    - Remove elements
    - Change elements

Lists

```
x = [1,2,3]                       # x is a list (note the square brackets)
y = list(("a", "b", "c", 6))  # y is also a list (note: (( )) )
z = list(["a", "b", "c", 6])  # z is also a list (note: ([ ]) )
```

Remark: *Element access and slicing works just as for tuples!*

```
print(x[0])
print(y[-1])
print(z[1:3])
```

```
1
6
['b', 'c']
```

Lists

Changing elements:

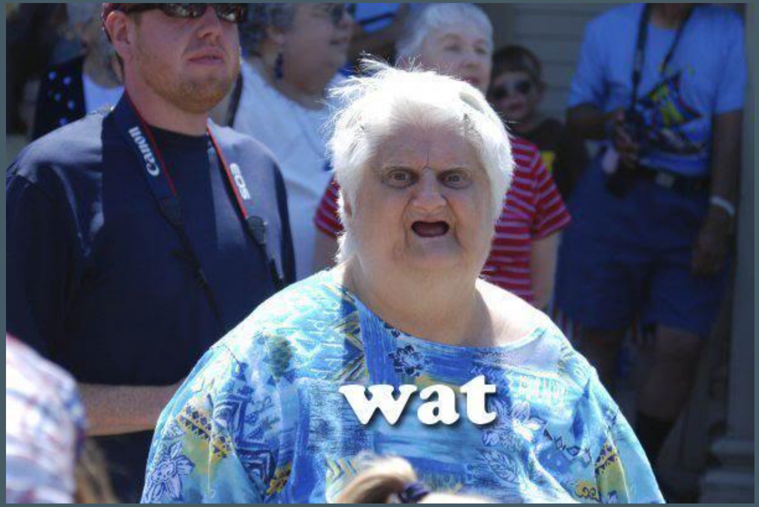
```
x = [1,2,3]
print(x)

x[2] = 0 # assign a different value to the third place, would not work with tuples!
print(x)
```

```
[1, 2, 3]
[1, 2, 0]
```

List Methods

Short intermezzo:

- All entities in Python are *objects*.
- Objects are so called *instances* of so called *classes* (= types!)
- A class bundles a bunch of functions ("methods") and variable names ("members")
- An object is created if we assign values to the class variables (oversimplified)



List Methods

```
x = list((1,2,3))
y = list(("a", "b"))
```

`x` and `y` are both *objects* of the `list` class (= type). They hold different values and are different objects.

The `list` class defines *methods* (functions) that can be called *using the object*

- *Methods* act on the object from which they were called
- They can change the object itself, or do something using the values stored in the respective object

List Methods

Example:

The class `list` has the method `append( elem )` (note: this is a function, thus `elem` is an argument)

- As the name says, it appends a new element to... to what?
- There is no argument asking for a list!
- That's because the method is *called from a list object*, and is appending the element to that list object!

List Methods

```python
x = [1, 2, 3]
print(x)

x.append(4)
print(x)
```

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

Take home message: By creating an *object* `x` of *class* (or type) `list` (via `[` `]`), Python automatically ties some *methods* to `x` for us to use.

(The blueprint (class definition) on how to do that already comes with Python.)

The notation to call methods is `objectname` `.` `methodname( argument(s) )`

List Methods

Important list methods:

- `pop(n)` - remove the element at index `n`
- `append(elem)` - append `elem` to the end of the list
- `extend(lst)` - append all elements in the *other list* `lst` to the end of the list
- `index(elem)` - returns the index of `elem` if it is in the list, otherwise raises an error

References

An important pitfall in Python: *references*

```
a = [3]
x = a
print(x)
a.append(4)
print(x)
```

What output do you expect?

References

An important pitfall in Python: *references*

```
a = [3]
x = a
print(x)
a.append(4)
print(x)    # yes, x, that's not a typo!
```

```
[3]
[3, 4]
```

...but why?

References

It is very desirable to *not* copy data if it is not necessary

- Lists and other objects can get very large, copying such objects takes time!

Thus, in Python, an expression like `x = a` by default creates a *reference* to `a` instead of copying all the data in `a` to `x`

`x` *is just **another name** for* `a`

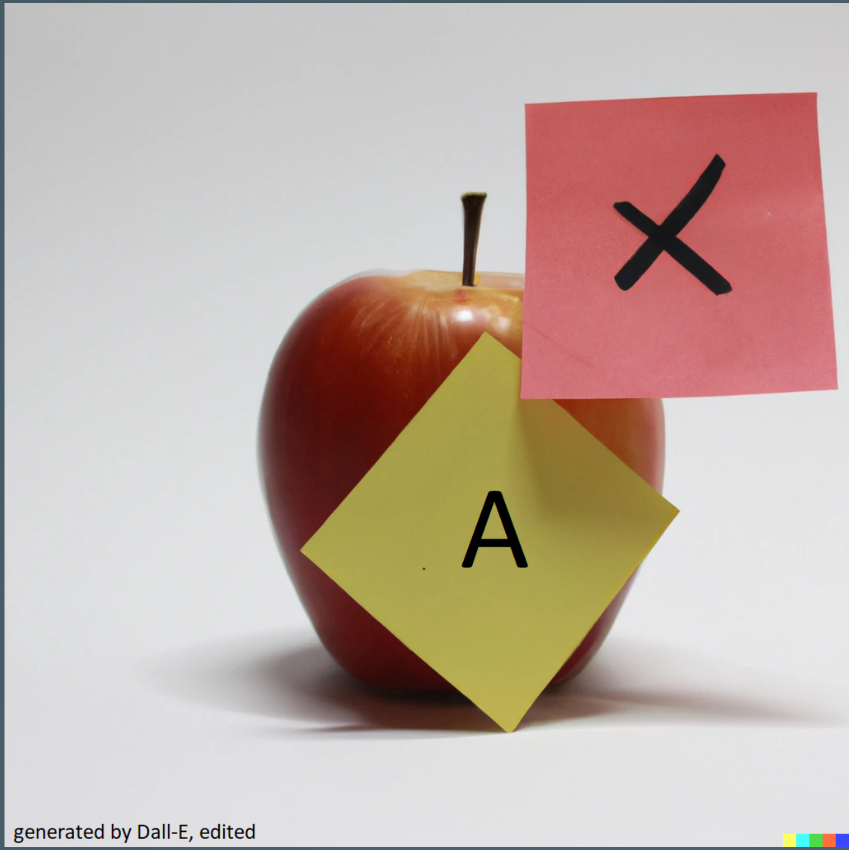
References

`x` *is just **another name** for* `a`

Imagine you have an apple.
This apple is an object (of the "class" *fruit of an apple tree*)

Take a post-it, write `x` on it, and stick it to the apple.
Your apple now has the name `x`.

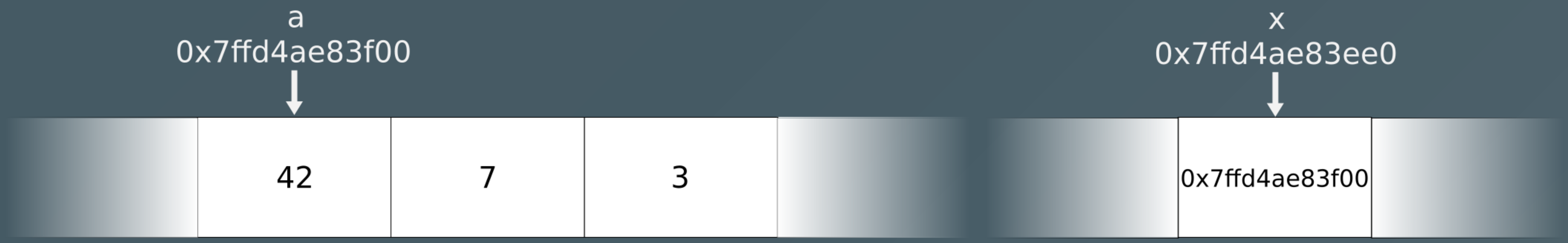
When you type `x = a`, it is the equivalent of taking another post-it, writing `a` on it and also sticking it to the apple.



References Under the Hood

```
a = [42, 7, 3]
x = a
```

Here is what happens in memory (not entirely true):



References

```
a = [1,2,3]
x = a       # x refers to the same actual data as a
a.append(4)
x.append(5) # both append to the same list
print(a)
```

```
[1, 2, 3, 4, 5]
```

References

```python
a = [1,2,3]
x = a        # x refers to the same actual data as a
a.append(4)
x.append(5) # both append to the same list
a = "foo"    # a now no longer refers to the list, but x still does
print(a)
print(x)
```

```
foo
[1, 2, 3, 4, 5]
```

References

Just remember:

When working with advanced data types like `list`, you usually don't get a real copy unless you ask for it!

```
a = [1,2,3]
x = a.copy()   # this creates a real copy
a.append(4)
x.append(5)
print(a)
print(x)
```

```
[1, 2, 3, 4]
[1, 2, 3, 5]
```

It's Your Turn!

- Please continue reading the notebook, read the "Lists" section

Do the exercise!

Exercise:

Take the shopping list below. Create a new empty list and append all the strings in `shopping_list` to it, so that you have a list with 9 string elements inside. **Do not copy and paste manually!** Finally, print the list and the size of the list.

Sets

Python also has the `set` type

As in mathematics, a `set` can hold each value only once!

```
x = set()
x.add(1)
x.add(2)
x.add(3)
x.add(1)
print(x)
```

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

Sets

Like lists:

- Can hold multiple values
- Are *dynamic*, i.e. you can
  - Append elements
  - Remove elements

But different:

- You cannot determine the index of an element in a set (elements are not ordered!)
- You cannot change an element inside a set
- Each value is only allowed once
- Some value types are not allowed (e.g. you cannot have nested sets), but `int`, `float`, `str`, `bool` and `tuple` are okay

Sets

As elements in a set are not ordered, you cannot use the `[ ]` operator to access an element

```
x = set([1,2,3])
print(x[0])
```

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

Sets

Add items:

```
x = set()
x.add( "foo" )
```

Add multiple items:

```
x.update([1,2,3])
```

Remove items:

```
x.remove( "foo" )  # raises an error if the element is not in the set
x.discard( "bar" ) # no error if the element is not in the set
```

Dictionaries

The last type for now is the *dictionary*, a collection of "key-value pairs"

- In lists and tuples, each element is at a certain known position and has an *index*
- In a dict, the elements are not ordered, but each has a unique *key* (usually a `str`) that is used to access a value

```python
# use curly braces to create a dict
my_dict = {
  'key1': 1,
  'key2': [1,2,3],
  'a set': set(['a','b']),
  'foo': 'bar'
}
# explicit construction is also possible
my_other_dict = dict()
```

Dictionaries

The keys of a dictionary behave much like a *set*

- Each key is unique
- The keys are not ordered, there are no numeric indices
- A key should always be a `str` (`list` etc. do not work!)

On the other hand, the *values* can be of any type, even another dict!

Dictionaries

The `[ ]` operator works for dicts, but not with numeric indices (as there are none), but with the *keys*

```
x = {'a': 1, 'b': 2, 'c': 3}
print(x['a'])
print(x['b'])
print(x['c'])
```

```
1
2
3
```

Dictionaries

You can also use the `[ ]` operator to add new key-value pairs or change the value of a present key

```
x = {'a': 1, 'b': 2, 'c': 3}
print(x)
x['d'] = 4
x['a'] = 0
print(x)
```

```
{'a': 1, 'b': 2, 'c': 3}
{'a': 0, 'b': 2, 'c': 3, 'd': 4}
```

Dictionaries

Methods

Use `pop( key )` to remove `key` and its associated value from the dict

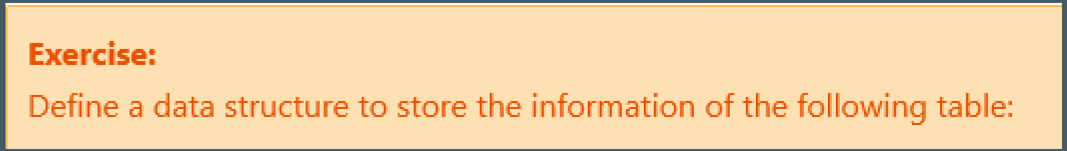
Use `copy()` to get a real copy of the dict (and not just a reference)

Use `keys()` to get an object containing all the keys in a dict

It's Your Turn!

- Please continue reading the notebook, read the "Lists" section

Do the exercise!



Exercise:

Define a data structure to store the information of the following table:

Go to our Moodle page (https://moodle.uni-greifswald.de/course/view.php?id=9565) and take the second quiz! ("Quiz 2 - Container Types")

Summary

You learned

- What a variable is and how it works
- The most important data types in Python:
    - `int`, `float`, `bool`, `str`
    - `tuple`, `list`
    - `set`, `dict`
- Basic arithmetic operations (`+`, `-`, `*`, `/`, ...)
- The `print()` function
- Brief introduction to functions

Next Up

The stuff that allows us to write *real* programs in Python:

- Control flow (if-then-else)
- Loops (repeat stuff)