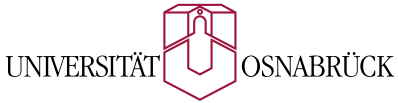
Week 9: Object Oriented Programming

Basic Programming in Python

Katharina Groß, Martin Pömsl, Sören Selbach

June 5, 2019



Week 9: Object Oriented Programming
Basic Programming in Python
Katharina Groß, Martin Pömsl, Sören Selbach
June 5, 2019
UNIVERSITÄT OSNABRÜCK

1 Recap
- Standard Library
- Functions

2 Objects
- What is an Object?
- Attributes & Methods
- Variables Revisited
- String Representation

3 OOP
- Motivation
- User-Defined Classes
- Properties
- Inheritance

4 Homework

The Standard Library

Python's **standard library** is a collection of modules included in most Python installations

- `math`: mathematical functions
- `random`: generate (pseudo-)random numbers
- `copy`: create shallow and deep copies of objects
- `time`: access the system's clock
- `os` & `shutil`: operating system dependent actions like copying files, manipulating paths etc.
- `sys`: technical things related to the Python interpreter, e.g. accessing command line arguments

The Standard Library
Python's **standard library** is a collection of modules included in most Python installations
- `math`: mathematical functions
- `random`: generate (pseudo-)random numbers
- `copy`: create shallow and deep copies of objects
- `time`: access the system's clock
- `os` & `shutil`: operating system dependent actions like copying files, manipulating paths etc.
- `sys`: technical things related to the Python interpreter, e.g. accessing command line arguments

As always, there are many more, and they each have their own documentation:

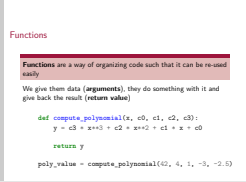
https://docs.python.org/3/library/

Functions

Functions are a way of organizing code such that it can be re-used easily

We give them data (**arguments**), they do something with it and give back the result (**return value**)

```
def compute_polynomial(x, c0, c1, c2, c3):
    y = c3 * x**3 + c2 * x**2 + c1 * x + c0

    return y

poly_value = compute_polynomial(42, 4, 1, -3, -2.5)
```

When a function is *defined* (with the `def` keyword), none of the code is actually executed! We only define what happends when the function is `called`. This can potentially be many times over the course of a program.

Section 2

Objects

What is an Object?

An object is the fundamental data structure of Python

- in Python, everything is an object:
  - lists, tuples, dictionaries etc.
  - strings
  - ints, floats
  - even functions!

What is an Object?
An object is the fundamental data structure of Python
in Python, everything is an object:
lists, tuples, dictionaries etc.
strings
ints, floats
even functions!

Type

Every object belongs to a type

```
>>> type([1, 2, 3])
<class 'list'>

>>> type(42)
<class 'int'>

>>> type(some_function)
<class 'function'>
```


Type
Every object belongs to a type
>>> type([1, 2, 3])
<class 'list'>
>>> type(42)
<class 'int'>
>>> type(some_function)
<class 'function'>

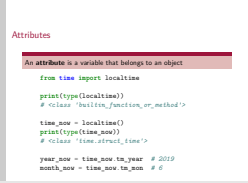
Objects always are *instances* of a class/type. There can be many instances of the same type - if you have three lists in your program, you have three instances of the type list.

In Python, the term *class* is mostly equivalent to *type*. "What is the type of this object" is equivalent to "What is the class of this object".
However, *class* is mostly used when talking about *user-defined* classes/-types (more on that later), while *type* is used when talking about *built-in* types/classes.

Attributes

An **attribute** is a variable that belongs to an object

```
from time import localtime

print(type(localtime))
# <class 'builtin_function_or_method'>

time_now = localtime()
print(type(time_now))
# <class 'time.struct_time'>

year_now = time_now.tm_year  # 2019
month_now = time_now.tm_mon  # 6
```

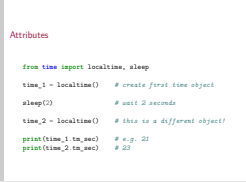



The **function** `localtime` returns an **object** of type `time.struct_time`. This object is stored in the **variable** `time_now` and has the **attributes** `tm_year` and `tm_month`.

Attributes of an object can be accessed by writing `object_name.attribute_name`

Attributes

```
from time import localtime, sleep

time_1 = localtime()    # create first time object

sleep(2)                # wait 2 seconds

time_2 = localtime()    # this is a different object!

print(time_1.tm_sec)    # e.g. 21
print(time_2.tm_sec)    # 23
```

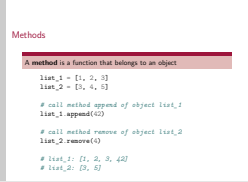



In general, the attributes with the same name but of different objects have different values!

Methods

A **method** is a function that belongs to an object

```
list_1 = [1, 2, 3]
list_2 = [3, 4, 5]

# call method append of object list_1
list_1.append(42)

# call method remove of object list_2
list_2.remove(4)

# list_1: [1, 2, 3, 42]
# list_2: [3, 5]
```

Just like attributes, methods know which object they belong to.

Variables Revisited

So far, we have treated variables like **containers** that store **values**
With this idea, we cannot explain this behavior:

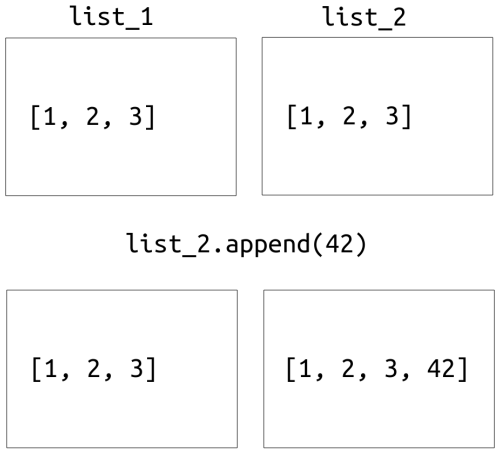
```
list_1 = [1, 2, 3]
list_2 = list_1

list_2.append(42)

# list_1: [1, 2, 3, 42]
# list_2: [1, 2, 3, 42]
```

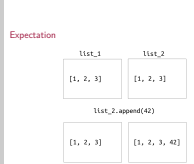

Variables Revisited

So far, we have treated variables like **containers** that store **values**
With this idea, we cannot explain this behavior:

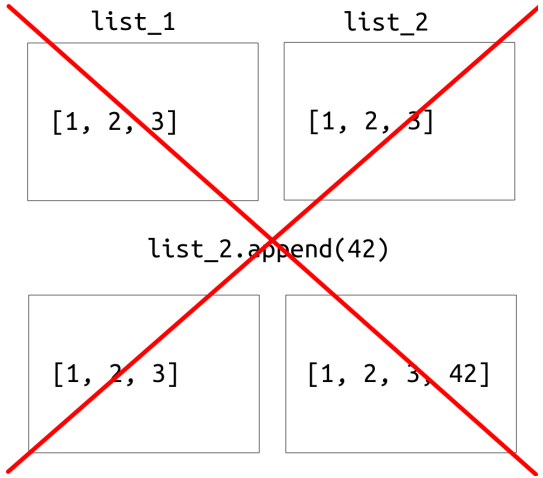
```
list_1 = [1, 2, 3]
list_2 = list_1

list_2.append(42)

# list_1: [1, 2, 3, 42]
# list_2: [1, 2, 3, 42]
```

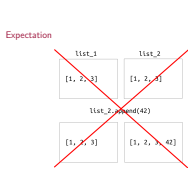
Expectation





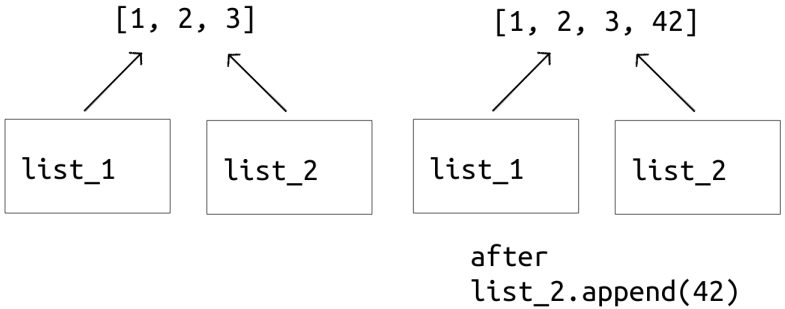
Expectation

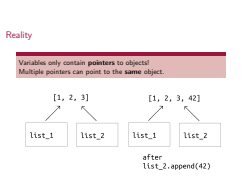




Reality

Variables only contain **pointers** to objects!
Multiple pointers can point to the **same** object.





This means that the only thing that is stored "in" a variable is an address of an object that is somewhere else in the computer's memory.

Note however, that if we create two identical lists like this:

```
list_1 = [1, 2, 3]
list_2 = [1, 2, 3]
```

... they are **not** the same object! You can also use the copy module to create copies of objects.

This only makes a difference when working with *mutable* objects such as lists or dictionaries.

String Representation

Some object types have an intuitive way to display them as text:

- `float`/`int`: use decimal notation
- `list`: use `[element1, element2, ...]`
- `time.struct_time`: `time.struct_time(tm_year=2019, tm_mon=6, ...)`

Others do not. In this case, you get a generic string:

- `<enumerate object at 0x7f38491d75a0>`
- `<function test at 0x7f384941b400>`

String Representation
Some object types have an intuitive way to display them as text:
float/int: use decimal notation
list: use [element1, element2, ...]
time.struct_time: time.struct_time(tm_year=2019, tm_mon=6, ...)
Others do not. In this case, you get a generic string:
<enumerate object at 0x7f38491d75a0>
<function test at 0x7f384941b400>

Example: Counter

The `collections` module provides the class `Counter`, which can count elements of lists, strings etc.:

```
from collections import Counter

some_string = "abccbcabcbcbbbabcbbbbab"

char_counter = Counter(some_string)
```

Now we have an **object** of type `Counter`

Example: Counter

The collections module provides the class Counter, which can count elements of lists, strings etc.:

from collections import Counter

some_string = "abccbcabcbcbbbabcbbbbab"

char_counter = Counter(some_string)

Now we have an object of type Counter

Example: Counter

This Counter object has methods we can access:

```
from collections import Counter

some_string = "abccbcabcbcbbbabcbbbbab"

char_counter = Counter(some_string)

print(char_counter.most_common(2))
# [('b', 13), ('c', 6)]
```


Example: Counter

This Counter object has methods we can access:

from collections import Counter

some_string = "abccbcabcbcbbbabcbbbbab"

char_counter = Counter(some_string)

print(char_counter.most_common(2))
# [('b', 13), ('c', 6)]

Section 3

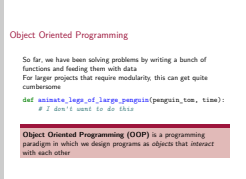
OOP

Object Oriented Programming

So far, we have been solving problems by writing a bunch of functions and feeding them with data
For larger projects that require modularity, this can get quite cumbersome

```
def animate_legs_of_large_penguin(penguin_tom, time):
    # I don't want to do this
```

Object Oriented Programming (OOP) is a programming paradigm in which we design programs as *objects* that *interact* with each other



Imagine we want to simulate a Zoo and want to visually animate the legs of all the animals. The logic for animating the legs of a Penguin is probably very different from animating the legs of an Elephant, and so we need different functions for that.

The problem here lies in how we organize these functions. We could have very long and detailed names like `animate_legs_of_large_penguin`, but this easily gets convoluted.

OOP aims to give programs more structure by grouping semantically related pieces of data and methods into **objects**.

Object Oriented Programming

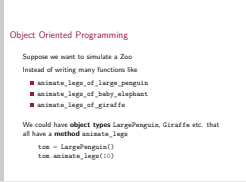
Suppose we want to simulate a Zoo

Instead of writing many functions like

- `animate_legs_of_large_penguin`
- `animate_legs_of_baby_elephant`
- `animate_legs_of_giraffe`

We could have **object types** `LargePenguin`, `Giraffe` etc. that all have a **method** `animate_legs`

```
tom = LargePenguin()
tom.animate_legs(10)
```

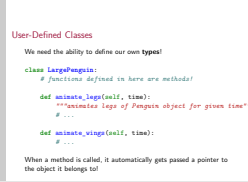



User-Defined Classes

We need the ability to define our own **types**!

```
class LargePenguin:
    # functions defined in here are methods!

    def animate_legs(self, time):
        """animates legs of Penguin object for given time"
        # ...

    def animate_wings(self, time):
        # ...
```

When a method is called, it automatically gets passed a pointer to the object it belongs to!



Note that the *naming convention* for user-defined classes is UpperCamelCase. Like functions, they also get their own **docstring** at the very top!

The Argument self: Remember how we said in the beginning that a method *knows* which object it belongs to? This should be weird to you, because we only define **one** method for **all** LargePenguins. How can it know on **which** LargePenguin it is called, as there could be many different instances?

Answer: When a method is called, it automatically gets passed a pointer to the object it belongs to! This is always the **first positional argument**. By convention, this is almost always called `self`, and it needs to be accepted in every (normal) method! We say a method is **bound** to an object.

User-Defined Classes

What about attributes? They can be defined similarly to variables:

```
# create LargePenguin object
tom = LargePenguin()

# give it the attribute 'age'
tom.age = 11
tom.fav_food = "fish"
```

99% of the time we want to define attributes **immediately after we create the object**

User-Defined Classes

What about attributes? They can be defined similarly to variables:

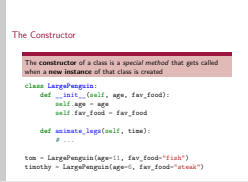
```
# create LargePenguin object
tom = LargePenguin()

# give it the attribute 'age'
tom.age = 11
tom.fav_food = "fish"
```

99% of the time we want to define attributes **immediately after we create the object**

The Constructor

The **constructor** of a class is a *special method* that gets called when a **new instance** of that class is created

```
class LargePenguin:
    def __init__(self, age, fav_food):
        self.age = age
        self.fav_food = fav_food

    def animate_legs(self, time):
        # ...

tom = LargePenguin(age=11, fav_food="fish")
timothy = LargePenguin(age=6, fav_food="steak")
```

The constructor **must** be named `__init__(self, ...)`!

Now `tom` is a `LargePenguin` object with attributes `tom.age == 11` and `tom.fav_food == "fish"`. `timothy` is also a `LargePenguin` object, but has attributes `timothy.age == 6` and `timothy.fav_food == "steak"`. Both have the method `animate_legs`.

Dunder Methods

The constructor is part of a family of special methods, called dunder-methods (double underscore)

They all do special things, depending on their name:

- `__init__`: contructor - gets called on object creation
- `__str__`: defines how objects of this type are converted to strings
- `__add__`: defines how objects of this type work with the "+" operator

There are of course many, many more.

Dunder Methods
The constructor is part of a family of special methods, called dunder-methods (double underscore)
They all do special things, depending on their name:
__init__: contructor - gets called on object creation
__str__: defines how objects of this type are converted to strings
__add__: defines how objects of this type work with the "+" operator
There are of course many, many more.

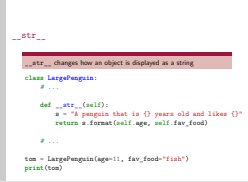
Because they affect the program without ever being explicitly called, dunder-methods are also often called *magic methods*.

Here you can find a complete list of all definable dunder-methods: https://docs.python.org/3/reference/datamodel.html#special-method-names

__str__

__str__ changes how an object is displayed as a string

```
class LargePenguin:
    # ...

    def __str__(self):
        s = "A penguin that is {} years old and likes {}"
        return s.format(self.age, self.fav_food)

    # ...

tom = LargePenguin(age=11, fav_food="fish")
print(tom)
```

__str__

Output without `__str__`:

```
<__main__.LargePenguin object at 0x7f384abebbe0>
```

Output with `__str__`:

```
A penguin that is 11 years old and likes fish
```


__str__
Output without __str__:
<__main__.LargePenguin object at 0x7f384abebbe0>
Output with __str__:
A penguin that is 11 years old and likes fish

Private Attributes

Sometimes we want to have **private** attributes (i.e. ones that are only accessible by the object they belong to)

This does not exist in Python.

Instead, this is handled by yet another **convention**:

```
class MyClass:
    def __init__(self):
        # this is marked as private with the _
        self._some_attr = 42
```


Private Attributes
Sometimes we want to have private attributes (i.e. ones that are only accessible by the object they belong to)
This does not exist in Python.
Instead, this is handled by yet another convention:
class MyClass:
def __init__(self):
# this is marked as private with the _
self._some_attr = 42

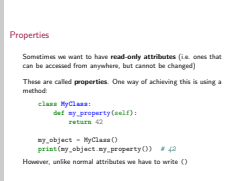
Properties

Sometimes we want to have **read-only attributes** (i.e. ones that can be accessed from anywhere, but cannot be changed)

These are called **properties**. One way of achieving this is using a method:

```
class MyClass:
    def my_property(self):
        return 42


my_object = MyClass()
print(my_object.my_property())  # 42
```

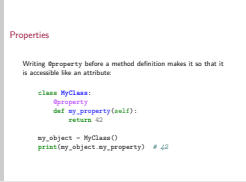
However, unlike normal attributes we have to write `()`



Properties

Writing `@property` before a method definition makes it so that it is accessible like an attribute:

```
class MyClass:
    @property
    def my_property(self):
        return 42

my_object = MyClass()
print(my_object.my_property)  # 42
```

`@property` is a **decorator**. Decorators modify the behavior of functions and methods. Again, there are more predefined ones, and you can define your own. We will not go into when and how to use them, as we don't have the time for that, but if you are running out of new things to learn, they can be useful.

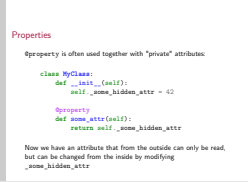
http://book.pythontips.com/en/latest/decorators.html

Properties

`@property` is often used together with "private" attributes:

```
class MyClass:
    def __init__(self):
        self._some_hidden_attr = 42

    @property
    def some_attr(self):
        return self._some_hidden_attr
```

Now we have an attribute that from the outside can only be read, but can be changed from the inside by modifying `_some_hidden_attr`



Again, note that technically, you still **can** change `_some_hidden_attr` from the outside - but you really should not. Whenever you are using `some_object._some_property` that starts with an `_`, you are using someone's code in an unintended way, which has the potential to break things.

Properties

```
>>> my_object = MyClass()
>>> my_object.some_attr
42
>>> my_object.some_attr = 43
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```


Properties

>>> my_object = MyClass()
>>> my_object.some_attr
42
>>> my_object.some_attr = 43
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: can't set attribute

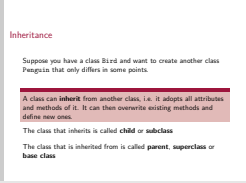
Inheritance

Suppose you have a class `Bird` and want to create another class `Penguin` that only differs in some points.

A class can **inherit** from another class, i.e. it adopts all attributes and methods of it. It can then overwrite existing methods and define new ones.

The class that inherits is called **child** or **subclass**

The class that is inherited from is called **parent**, **superclass** or **base class**



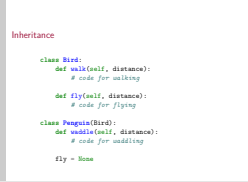
Think of inheritance as making some concept more specialized. Think of the base class `Phone`, which has a method `call`. A `SmartPhone` inherits from `Phone` because it is a specialized version of a phone. It might have the method `send_message` in addition. An old `RotaryPhone` also inherits from `Phone`, but does not inherit from `SmartPhone`.

In Python, everything automatically inherits from `object`. *Exceptions* are another nice example: Every Exception inherits from `BaseException`, i.e. `SyntaxError`. `IndentationError` in turn inherits from `SyntaxError`. Here is the hierarchy again:

https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Inheritance

```
class Bird:
    def walk(self, distance):
        # code for walking

    def fly(self, distance):
        # code for flying

class Penguin(Bird):
    def waddle(self, distance):
        # code for waddling

    fly = None
```

`Penguin` inherits from `Bird`, which means it gets the methods `walk` and `fly` without having to re-implement them. Now there are two things going on:

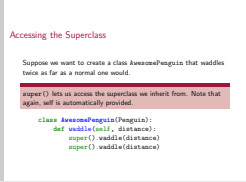
1. Penguins can waddle, while normal birds cannot, and so we add the method `waddle`.
2. Penguins cannot fly, and so we overwrite the method `fly`. There are a couple of ways for doing this - we could for instance re-define it in `Penguin` and have it do nothing. The convention is to just set the entire method to `None` (so it is not even a method afterwards). This way, when you try to call `some_penguin.fly(10)` you get a `TypeError`

Accessing the Superclass

Suppose we want to create a class `AwesomePenguin` that waddles twice as far as a normal one would.

`super()` lets us access the superclass we inherit from. Note that again, self is automatically provided.

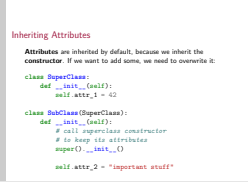
```
class AwesomePenguin(Penguin):
    def waddle(self, distance):
        super().waddle(distance)
        super().waddle(distance)
```

Inheriting Attributes

Attributes are inherited by default, because we inherit the **constructor**. If we want to add some, we need to overwrite it:

```
class SuperClass:
    def __init__(self):
        self.attr_1 = 42

class SubClass(SuperClass):
    def __init__(self):
        # call superclass constructor
        # to keep its attributes
        super().__init__()

        self.attr_2 = "important stuff"
```

Reading

This is a good (and rather short) article to recap your understanding of objects and types in Python:

`https://eli.thegreenplace.net/2012/03/30/python-objects-types-classes-and-instances-a-glossary`

Reading
This is a good (and rather short) article to recap your understanding of objects and types in Python:
https://eli.thegreenplace.net/2012/03/30/python-objects-types-classes-and-instances-a-glossary

Section 4

Homework