Python core language part 3

Content

You will learn:

- functions
- classes

Functions

A **function** is a block of code, which only runs when it is called.

It allows to divide your code into useful blocks.

Syntax:

```
def function_name(input_parameter):
    statement
    […]
    return output_parameter
```

In [ ]:
```
def foo():
    pass
```

In [ ]:
```
foo()
```

In [ ]:
```
# Example
def plus(a, b):
    print(a)
    print(b)
    return a + b
```

In [ ]:
```
bar = plus(3,4)
```

```
3
4
```

In [ ]:
```
bar
```

Out[ ]:
```
7
```

In [ ]:
```
def gauss(n):
    return (n*(n+1)/2)
```

In [ ]:
```
gauss(100)
```

Out[ ]:
```
5050.0
```

In [ ]:
```
def gaus(x):
    g = (x*(x+1))/2
    return g
```

In [ ]:
```
gaus(100)
```

Out[ ]:
```
5050.0
```

In [ ]:
```
def gauss2(n):
    if 1 == n:
        return n
    else:
        return n+gauss2(n-1)
```

In [ ]:
```
gauss2(100)
```

Out[ ]:
```
5050
```

gauss2(100)

return 100 + 4950

gauss2(99)

return 99 + gauss2(98)

gauss2(2)

return 2 + 1

gauss2(1)

return 1

In [ ]:
```
def bar(a,b):
    a_new= a+b
    b_new= a*b
    c_new= a_new + b_new
    return a_new,b_new,c_new
```

In [ ]:
```
baz1 = bar(10,20)
print(baz1[0])
```

```
30
```

In [ ]:
```
baz1,_,_ = bar(10,20)
print(baz1)
```

```
30
```

Excercise

1. Implement the Gauss formula as a function.
2. Compare the results with the for loop for different cases.

Classes

A python class is a collection of data and functions and can act as a blueprint, from which any number of instances can be created from. The class represents a new type and each of its instantiated objects has the same type. The functionality of python's class concept is geared to C++ and therefore offers all standard class concepts like inheritance. Since inheritance is a rather large topic, we will focus on the basic concepts; more information can be found in the official documentation of python.

Oversimplified a class offers a new namespace, by which all attributes of a class can be accessed and are separated from other variables or functions, which have the same name. All attributes (methods or variables) of a class are bound to the class or their objects. An attribut might be either a data attribute or a method.

Very minimal working example

In [ ]:
```
class VeryMinimalExample:
    pass

foo = VeryMinimalExample()
```

In [ ]:
```
type(VeryMinimalExample)
```

Out[ ]:
```
type
```

Minimal working example, which does something

In [ ]:
```
class MinimalExample:
    """Minimal working class"""
    foo = 42

    def baz():
        print("Hallo Welt")

    def bar(self):
        print("Hello World")
```

In [ ]:
```
def baz():
    print("Bonjour")
```

In [ ]:
```
baz()
```

```
Bonjour
```

Using a class

Classes can be used for two things:

1. Access *global* class attributes
2. Use it as a blueprint to instantiate a class object

In [ ]:
```
print(MinimalExample.foo)
```

```
17
```

Class attributes can be read from as well as be written to.

In [ ]:
```
MinimalExample.foo = 17
print(MinimalExample.foo)
```

```
17
```

In [ ]:
```
MinimalExample.baz()
```

```
Hallo Welt
```

In [ ]:
```
baz()
```

```
Bonjour
```

`bar(self)` takes the an argument `self`, which represents the instantiated class object. It's convention and good practice to name the first argument of a class method `self`. It can be named as you want (even though this is considered a bad style), its only important that it is the first argument. Via `self` the methods can access the object's attributes, on which the method is executed on. In the next section you will see how to instantiate such an object. Therefore, to call `bar()`, an class instance is needed.

In [ ]:
```
MinimalExample.bar()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
Input In [48], in <cell line: 1>()
----> 1 MinimalExample.bar()

TypeError: bar() missing 1 required positional argument: 'self'
```

Instantiate an object

To create an object simply assume that the class is a function without parameters (in this case. It is possible, that the constructor requires you to pass arguments).

In [ ]:
```
myObj = MinimalExample()
```

In [ ]:
```
print(myObj.foo)
```

```
17
```

In [ ]:
```
myObj.baz()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
Input In [51], in <cell line: 1>()
----> 1 myObj.baz()

TypeError: baz() takes 0 positional arguments but 1 was given
```

In [ ]:
```
#MinimalExample.baz()
myObj.bar()
```

Out[ ]:
```
'Hallo'
```

Like class attributes you can change attributes of an instance as well.

In [ ]:
```
def hallo():
    return "Hallo"
```

In [ ]:
```
hallo
```

Out[ ]:
```
<function __main__.hallo()>
```

In [ ]:
```
myObj.bar = hallo
myObj.bar()
```

Out[ ]:
```
'Hallo'
```

In [ ]:
```
myObj2 = MinimalExample()
```

In [ ]:
```
MinimalExample.__dict__
```

Out[ ]:
```
mappingproxy({'__module__': '__main__',
              '__doc__': 'Minimal working class',
              'foo': 42,
              'baz': <function __main__.MinimalExample.baz()>,
              'bar': <function __main__.MinimalExample.bar(self)>,
              '__dict__': <attribute '__dict__' of 'MinimalExample' objec
ts>,
              '__weakref__': <attribute '__weakref__' of 'MinimalExample'
objects>})
```

In [ ]:
```
myObj2.bar()
```

```
Hello World
```

In [ ]:
```
myObj2.foo
```

Out[ ]:
```
42
```

In [ ]:
```
myObj.foo = 1337
print(myObj.foo)
```

```
1337
```

In [ ]:
```
print(MinimalExample.foo)
```

```
42
```

Explicit constructor

Class instantiations are by default empty objects. A constructor (`__init__`) can be implemented to define the initial state of the object. `self` has no special meaning in python but is used in python to denote that a function or attribute belongs to a specific object and not the general class. Each class has a built-in `__init__(self)` function, either explicit (overridden) or implicit (original built-in). It is executed by calling the class like a function, which executes the `__init__(self)` function. It corresponds to the constructor method in C++. Like in C++ there is also a destructor, which is in Python `__del__(self)` . Since Python has a garbage collector, there are fewer use cases to make use of the destructor.

In [ ]:
```
class MediumExample:
    """Class example with data initialisation"""

    def __init__(self, x):
        self.x = x
        self.y = 10
```

In [ ]:
```
print(MediumExample.y)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call las
Input In [85], in <cell line: 1>()
----> 1 print(MediumExample.y)

AttributeError: type object 'MediumExample' has no attribute 'y'
```

In [ ]:
```
myObj2 = MediumExample(1)
```

In [ ]:
```
print(myObj2.x)
print(myObj2.y)
```

```
1
10
```

New object attributes can be created on the fly and do not interfere with other existing objects or the class itself.

In [ ]:
```
print(myObj2.baz)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call las
Input In [88], in <cell line: 1>()
----> 1 print(myObj2.baz)

AttributeError: 'MediumExample' object has no attribute 'baz'
```

In [ ]:
```
myObj2.baz=100
print(myObj2.baz)
```

```
100
```

New methods can be created as well.

In [ ]:
```
def add(a,b):
    return a+b
```

In [ ]:
```
myObj2.new_method = add
```

In [ ]:
```
myObj2.new_method(20,30)
```

Out[ ]:
```
50
```

Attributes can also be deleted on-the-fly with the `del` statement.

In [ ]:
```
del myObj2.baz
```

In [ ]:
```
print(myObj2.baz)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call las
Input In [94], in <cell line: 1>()
----> 1 print(myObj2.baz)

AttributeError: 'MediumExample' object has no attribute 'baz'
```

In [ ]:
```
print(myObj2)
```

```
<__main__.MediumExample object at 0x7fffd4674940>
```

Demo

We create a class, which represents pixel. The class should satisfy the following requirements:

1. Three attributes, which represent the RGB channel (one attribute for the red value, one attribute for the green channel, one attribute for the blue channel). Those values shall be passed to the object during its creation. Make sure that only values between 0 and 255 are accepted: Values below zero should be set to zero, values above 255 to 255!
2. The class shall implement a method `__str__(self)`, which returns a string, which characterizes a pixel, i.e. if you print your pixel object (like `print(Pixel(50,100,150))`) it shall return a descriptive string.

In [ ]:
```
# Basic skeleton
class Pixel():
    def __init__(self, r, g, b):
        pass
```

Pass initial arguments of pixel to class

In [ ]:
```
# Pass arguments
class Pixel():
    def __init__(self, r, g, b):
        self.r = r
        self.g = g
        self.b = b
```

In [ ]:
```
test=Pixel(1,2,-3)
print(test.b)
```

```
-3
```

Perform sanity checks so that each value is in between 0 and 255. There are multiple possibilities how to do this, three ways are demonstrated. They differ regarding verbosity and lines of code.

In [ ]:
```
# Check passed arguments
class Pixel():
    def __init__(self, r, g, b):
        # channel r
        if r < 0:
            self.r = 0
        elif r > 255:
            self.r = 255
        else:
            self.r = r

        # channel g
        self.g = 0 if g < 0 else 255 if g > 255 else g

        # channel b
        self.b = min(255, max(0, b))
```

To check we create three pixels

In [ ]:
```
pixel1 = Pixel(10,20,30)
pixel2 = Pixel(-150,0,150)
pixel3 = Pixel(100,200,300)
print(pixel1)
print(pixel2)
print(pixel3)
```

```
<__main__.Pixel object at 0x7fffd4c34b50>
<__main__.Pixel object at 0x7fffd4c34400>
<__main__.Pixel object at 0x7fffd4c34340>
```

min(255, max(0, -10)) -> min(255,0) -> 0

min(255, max(0,300)) -> min(255, 300) -> 255

Not very expressive yet, therefore we setup a special method, so that `print` which values of the object to print

In [ ]:
```
# Implement __str__ method
class Pixel():
    def __init__(self, r, g, b):
        self.r = min(255, max(0, r))
        self.g = min(255, max(0, g))
        self.b = min(255, max(0, b))

    def __str__(self):
        return "Hello World"
```

In [ ]:
```
# don't forget to reinstance the objects, otherwise the objects following
pixel1 = Pixel(10,20,30)
pixel2 = Pixel(-150,0,150)
pixel3 = Pixel(100,200,300)
print(pixel1)
print(pixel2)
print(pixel3)
```

```
Hello World
Hello World
Hello World
```

In [ ]:
```
# Implement __str__ method
class Pixel():
    def __init__(self, r, g, b):
        self.r = min(255, max(0, r))
        self.g = min(255, max(0, g))
        self.b = min(255, max(0, b))

    def __str__(self):
        #return str(self.r) + " " + str(self.g) + " " + str(self.b)
        return f"R:{self.r}\tG:{self.g}\tB:{self.b}\tHex:#{self.r:02x}{se
```

In [ ]:
```
pixel4,test2 = Pixel(1,2,3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
Input In [127], in <cell line: 1>()
----> 1 pixel4,test2 = Pixel(1,2,3)

TypeError: __init__() should return None, not 'tuple'
```

In [ ]:
```
# don't forget to reinstance the objects, otherwise the objects following
pixel1 = Pixel(10,20,30)
pixel2 = Pixel(-150,0,150)
pixel3 = Pixel(100,200,300)
print(pixel1)
print(pixel2)
print(pixel3)
```

```
R:10    G:20    B:30    Hex:#0a141e
R:0     G:0     B:150   Hex:#000096
R:100   G:200   B:255   Hex:#64c8ff
```

In [ ]:
```
for i in [1,2,3]:
    print(i)
```

```
1
2
3
```

Special attributes

There are many special attributes, which allows special interaction with your objects. We will have a closer look on one example to get the idea of this concept.

An iterator represents a stream of data, which can be easily iterated. If your data structure supports iteration it has the advantage that it can easily be used in a `for` loop. An iterable object can can be iterated through its values by implementing `__iter__()` and `__next__()`.

- `__iter__()` returns the iteration object, which can be used to go through the values.
- `__next__()` returns the next object in the sequence.

To stop the iteration (necessary if the output is not created dynamically) you can raise an `StopIteration` exception. We will not cover exceptions here, so to prevent further output you can add a condition branch with `raise StopIteration` as soon as the last element is reached.

Collections are iterate-able, i.e. you can get an iterator object from them.

In [ ]:
```
fruits = ["Ananas", "Apple", "Banana"]
iterator = iter(fruits)
print(next(iterator))
next(iterator)
print(next(iterator))
print(next(iterator))
```

```
Ananas
Banana
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call las
Input In [132], in <cell line: 6>()
      4 next(iterator)
      5 print(next(iterator))
----> 6 print(next(iterator))

StopIteration:
```

A `for` loop makes use of an iterator.

In [ ]:
```
for fruit in fruits:
    print(fruit)
```

```
Ananas
Apple
Banana
```

Demo

Create a new collector class `PixelCollection`, which represents a data structure: it stores a list of single `Pixel` objects and allows to easily iterate them. You can decide if you want to pass a complete list during the creation of a `PixelCollection` object or if you implement a method, to add single `Pixel` objects one by one.

In [ ]:
```
# skeleton
class PixelCollection():
    def __init__(self, pixel_list):
        pass

    def __iter__(self):
        pass

    def __next__(self):
        pass
```

First save the initial pixel list and allow to add new pixels. We will ignore for this demo convenience features like accessing or removing objects and no sanity checks will be performed (e.g. make sure that passed objects really belong to `class Pixel`).

To keep things easier, we make sure that the constructor can be called without any list by using an optional argument `pixel_list=[]`. If nothing is passed, the empty list us used.

In [ ]:
```
# skeleton
class PixelCollection():
    def __init__(self, pixel_list=[]):
        self.collection = pixel_list

    def append(self, pixel):
        self.collection.append(pixel)
```

Now we assign a `PixelCollection` object to the variable `collection`, and add `pixel1`, `pixel2` and `pixel3` to the collection.

In [ ]:
```
collection = PixelCollection()
print(collection.collection)
```

```
[]
```

In [ ]:
```
for pixel in [pixel1, pixel2, pixel3]:
    collection.append(pixel)
print(collection.collection)
```

```
[<__main__.Pixel object at 0x7fffd4590460>, <__main__.Pixel object at 0x7
fffd45909d0>, <__main__.Pixel object at 0x7fffd4590fd0>]
```

Our collection is set up but the iteration with a `for` loop does not work yet.

In [ ]:
```
for pixel in collection:
    print(pixel)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
Input In [139], in <cell line: 1>()
----> 1 for pixel in collection:
      2     print(pixel)

TypeError: 'PixelCollection' object is not iterable
```

In [ ]:
```python
# define __iter__
class PixelCollection():
    def __init__(self, pixel_list=[]):
        self.collection = pixel_list

    def append(self, pixel):
        self.collection.append(pixel)

    def __iter__(self):
        return self
```

Define method to iterate the pixels in our collection. For this we need to remember the idnex of the current pixel

In [ ]:
```python
# define __next__
class PixelCollection():

    def __init__(self, pixel_list=[]):
        self.collection = pixel_list
        self.index = -1

    def append(self, pixel):
        self.collection.append(pixel)

    def __iter__(self):
        return self

    def __next__(self):
        self.index += 1
        return self.collection[self.index]
```

In [ ]:
```python
collection = PixelCollection([pixel1,pixel2,pixel3])
for pixel in collection:
    print(pixel)
```

```
R:10    G:20    B:30    Hex:#0a141e
R:0     G:0     B:150   Hex:#000096
R:100   G:200   B:255   Hex:#64c8ff
```

In [ ]:
```python
#collection.append(pixel4)
for pixel in collection:
    print(pixel)
```

```
R:10    G:20    B:30    Hex:#0a141e
R:0     G:0     B:150   Hex:#000096
R:100   G:200   B:255   Hex:#64c8ff
R:10    G:20    B:30    Hex:#0a141e
```

Almost there! It only be good, if our collection would not throw an error if its end is reached!

In [ ]:
```
# define throw StopIteration at the end
class PixelCollection():

    def __init__(self, pixel_list=[]):
        self.collection = pixel_list

    def append(self, pixel):
        self.collection.append(pixel)

    def __iter__(self):
        self.index=-1
        return self

    def __next__(self):
        self.index += 1
        if self.index >= len(self.collection):
            raise StopIteration
        return self.collection[self.index]
```

In [ ]:
```
collection = PixelCollection([pixel1,pixel2,pixel3])
```

In [ ]:
```
for pixel in collection:
    print(pixel)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
Input In [174], in <cell line: 1>()
----> 1 for pixel in collection:
      2     print(pixel)

TypeError: 'PixelCollection' object is not iterable
```

Now it works! Our design has definitely still some flaws but this is enough at the moment.

In [ ]:
```
pixel4 = Pixel(10,20,30)
```

In [ ]:
```
id(pixel1)
```

Out[ ]:
```
140736751085600
```

In [ ]:
```
pixel1.__hash__()
```

Out[ ]:
```
8796046942850
```

In [ ]:
```
id(pixel4)
```

Out[ ]:
```
140736762654880
```

In [ ]:
```
a = 37
```

In [ ]:
```
pixel4.__hash__()
```

Out[ ]:
```
8796047665930
```

In [ ]:
```
a = 42
```

In [ ]:
```
a == 42
```

Out[ ]:
```
False
```

In [ ]:
```
a
```

Out[ ]:
```
37
```

In [ ]:
```

```