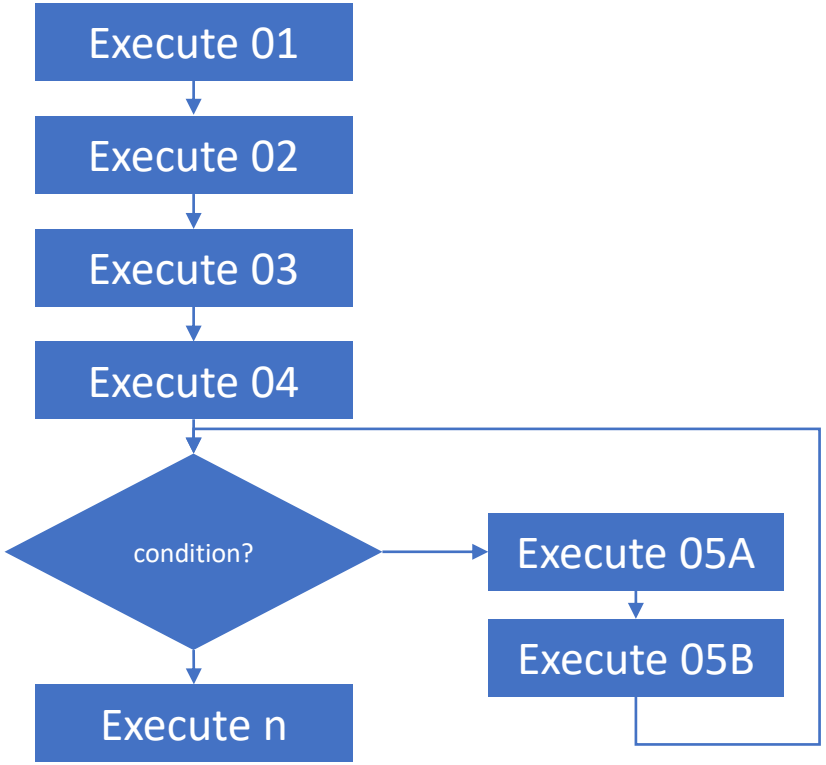
Object Oriented Programming

Object Oriented Programming

Procedural Programming

- sequential code
- sequence of instructions executed
- executed from top to bottom
- can have loops, iterations, …
- code blocks split up into functions

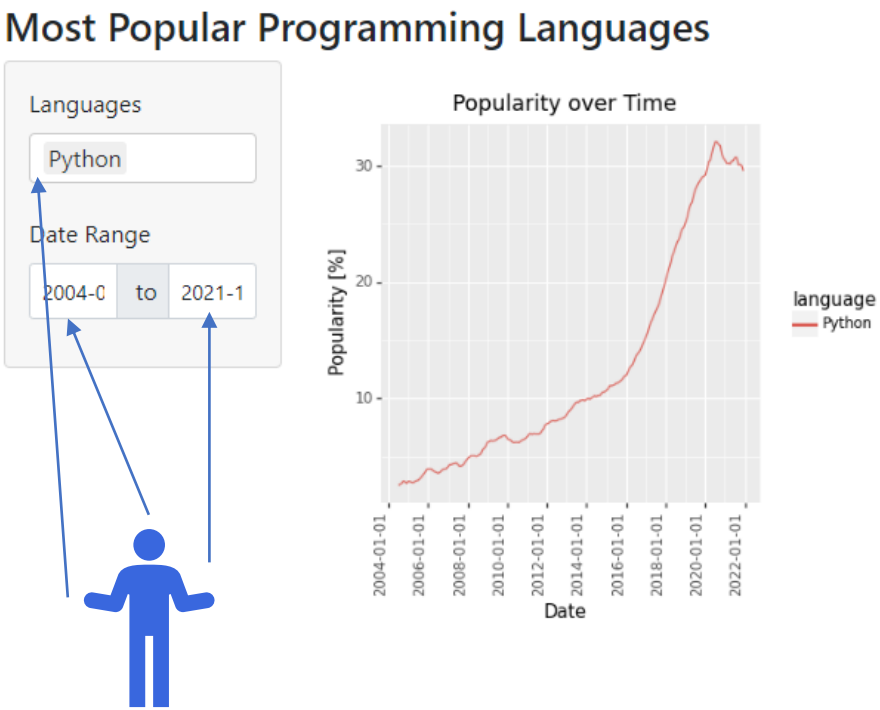
- program based on functions
- typical programming languages C, Basic, …



Object Oriented Programming

OOP introduction

- programming paradigm
- use objects to represent things
- objects don‘t need to be processed in certain order



User can interact at different places with app

Object Oriented Programming

Main Features of OOP

Classes

- construction manual (blueprint) to create objects (instances of class)

Class Instance

- an object of the class

Methods

- functionality to interact with instances

Object Oriented Programming

Class and Instances

class **Car**

name

max_speed

color

start()

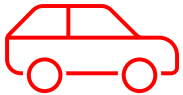
stop()

Legend

property

method

Instance



name = „family_car“

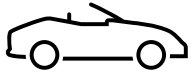
max_speed = 100

color =„red“

start()

stop()

Instance



name = „sports_car“

max_speed = 200

color =„black“

start()

stop()

Object Oriented Programming

Why to use OOP?

- Classes hold many properties and methods

```
my_name = 'Bert'
```

```
my_name.__class__
```

```
str
```

```
my_name.
```

```
★ replace
★ format
★ count
★ title
rindex
rindex
rfind
rfind
removesuffix
replace
removeprefix
removesuffix
```

```
my_list = list(range(10))
```

```
my_list.__class__
```

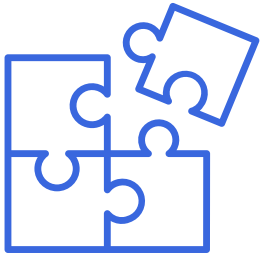
```
list
```

Object Oriented Programming

Why to use OOP?

simplify problems

Reuseable code pieces



Use smaller pieces of code

Object Oriented Programming

Creating a Class

- *class*…keyword reserved for creating classes
- *__init__*…function for initializing attributes of class
- *self*…reference to instance object

```
19  class Pet:
20      def __init__(self, name, species):
21          self.name = name
22          self.species = species
23
24      def hello(self):
25          print(f"Hello! My name is {self.name} and I am a {self.
            species}")
26
27  pet_one = Pet("Kiki", "dog")
28  pet_two = Pet("Bubbles", "cat")
```

- instances are created by calling the class and passing required parameters

Object Oriented Programming

Instances

- Instances are created by typing name of the class together with brackets and required parameters
- here: two new instances are created
- instance *pet_new* refers to memory address
- instance *pet_new2* seems to be identical to *pet_new,* but

```
pet_new == pet_new2
✓ 0.3s
False
```

- because, *pet_new2* is another instance (with the same attributes), but a different memory address

```
pet_new = Pet('Waldo', 'dog')
pet_new2 = Pet('Waldo', 'dog')
```

```
pet_new
✓ 0.4s
<__main__.Pet at 0x1de3c195210>
```

```
pet_new2
✓ 0.5s
<__main__.Pet at 0x1de3c196590>
```

Object Oriented Programming

Instance Attributes

- *self.name = name* creates a class-attribute called name and assigns value of parameter name
- attributes created in __init__ are instance attributes
- opposite: class-attribute is the same for all class instances

```
class Pet:
    is_human = False  # class attribute, identical for all
    instances

    def __init__(self, name):
        self.name = name
```

Object Oriented Programming

Instance Methods

- functions inside a class
- can only be called from instance of class

- .hello() method makes use of instance attributes, so it needs to get access – *self* is passed as parameter

```
class Pet:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def hello(self):
        print(f"Hello! My name is {self.name} and I am a {self.
        species}")
```

Object Oriented Programming

Class Inheritance

- create new classes (derived classes, descendants) based on existing classes (ancestors)
- derived class inherits all attributes and methods from ancestor class

ancestor class

descendant class

```
class Pet:...


class Dog(Pet):
    def __init__(self, name, breed):
        super().__init__(name)
        self.species = 'dog'
        self.breed = breed

    def hello(self):
        print(f"Hello! My name is {self.name}, I am a dog of
        type {self.breed}")
```