UNIVERSITÄT BONN

SPP 2363

OBJECT-ORIENTED-PROGRAMMING IN PYTHON

Overview

Part I – Object-Oriented Programming in Python

- Lecture (40min): getting to know the basics
- Tutorial (20min): hands-on session

Part II – Project Setup in Python

Part I

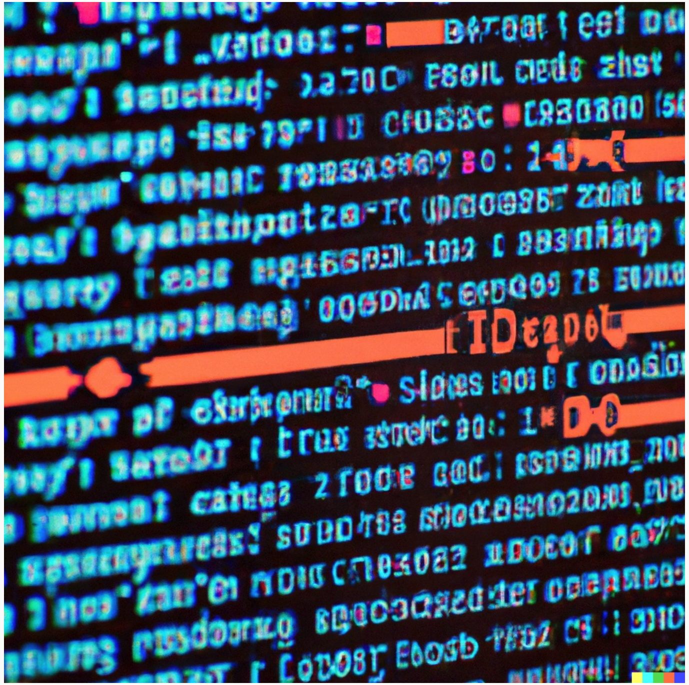
Object-Oriented-Programming in Python

Why do we write Computer Programs?

1) Perform calculations
2) Read, manipulate and store data
3) Simulate aspects of real-life

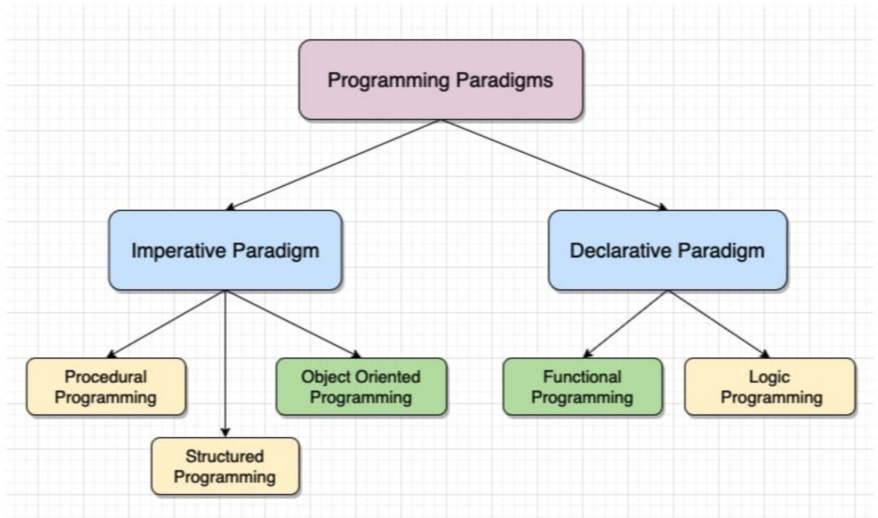
All programs will therefore contain :

- **Data**: store the state of a system
- **Methods**: change the state of a system



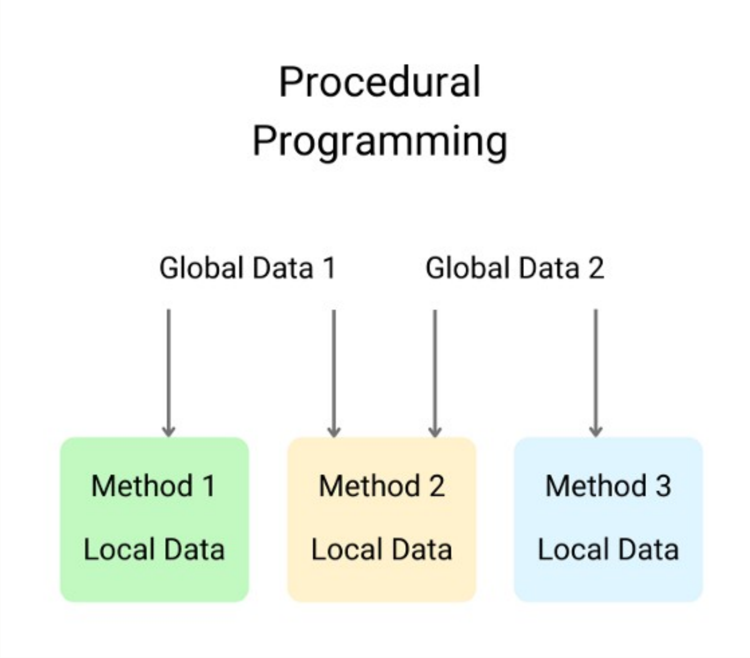
Programming Paradigms

- **Procedural**: step-wise procedures carried out successively
- **OOP**: model objects and how they interact with data



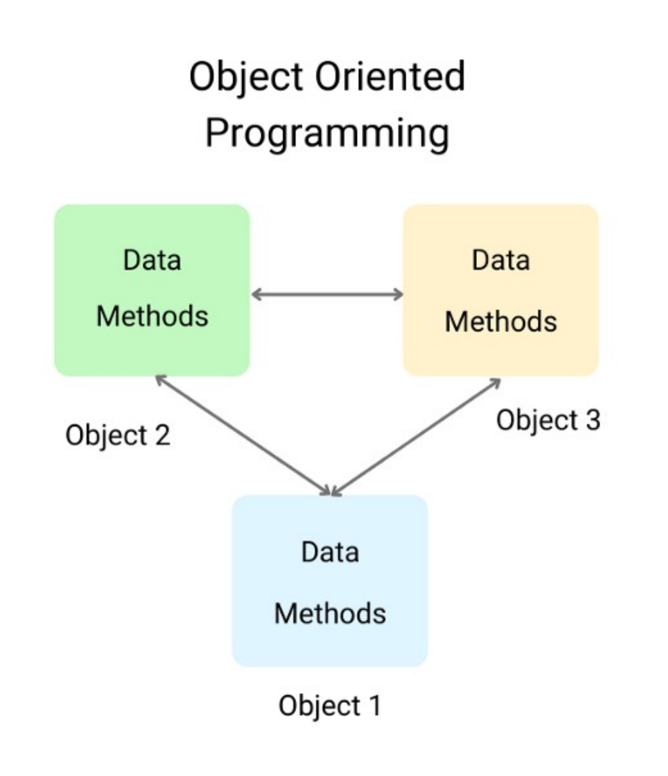
Procedural Programming

- **Global data**
- Only basic data types
  Primitives: int, float, str, bool
  Non-primitives: list, dict, etc.
- Methods have no internal state and (generally) operate on input only



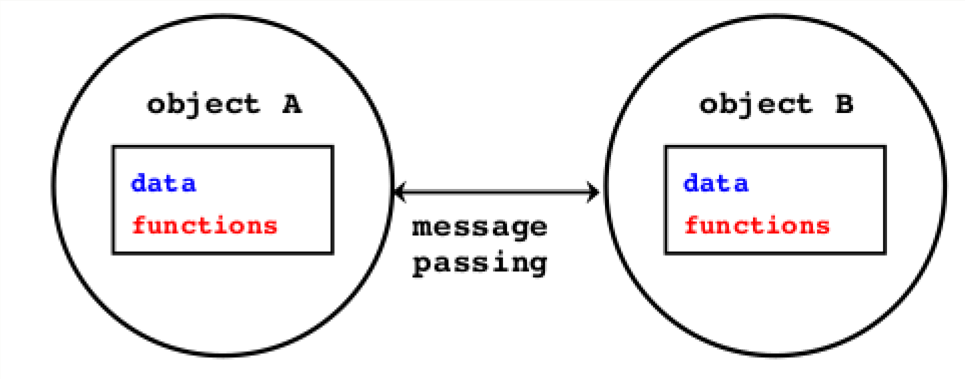
Object-Oriented Programming

- Extend data types by objects
- **Objects** have internal state and **can store data and methods**



But what is an object?

- An object is a **“bundle” of data and methods**
- Objects can interact with each other
- Objects allow to organize large programming tasks



But what is an object in Python?

In Python, *everything* is an object.

```python
# example object
class Cat:
    name: str
    age: int

    def make_sound(self):
        print("Meow.")
```

```python
# Assign an integer to a variable
x = 10

# Check the type of the variable
print(type(x))  # Output: <class 'int'>

# Check the attributes and methods of the integer object
print(dir(x))

# Output:
# ['__abs__', '__add__', ... ,
#  'bit_length', 'conjugate', 'to_bytes']

# Call a method of the integer object
print(x.bit_length())  # Output: 4
```

Handling an object

Access variables and methods via “.“

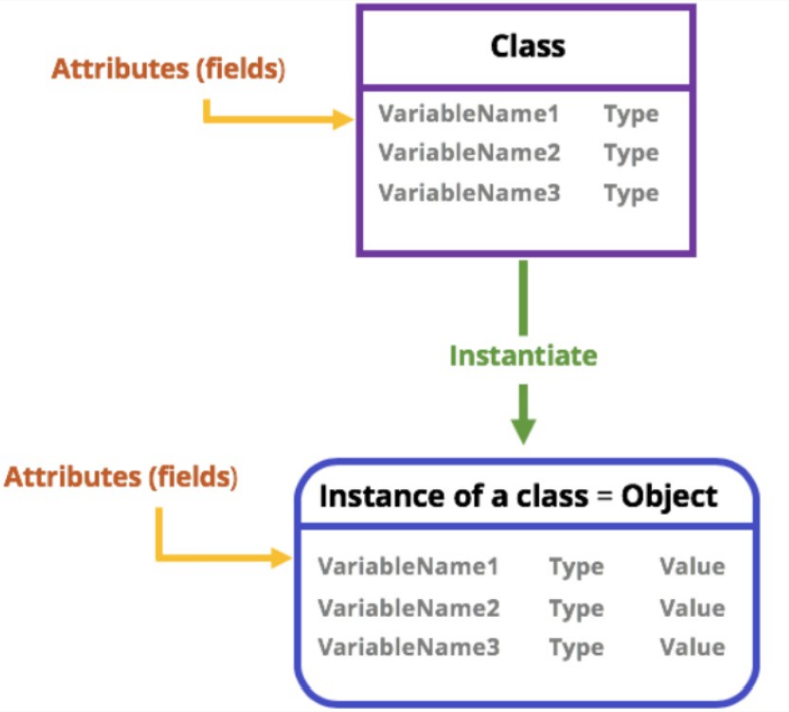
Self-reference via “self” keyword

```python
# example object
class Cat:
    name: str
    age: int

    def make_sound(self):
        print("Meow.")
```

```python
c = Cat()
c.age = 5 # acess variable
c.make_sound() # access method

c2 = Cat()
c2.age = 7

cats = [Cat() for i in range(100)]
```

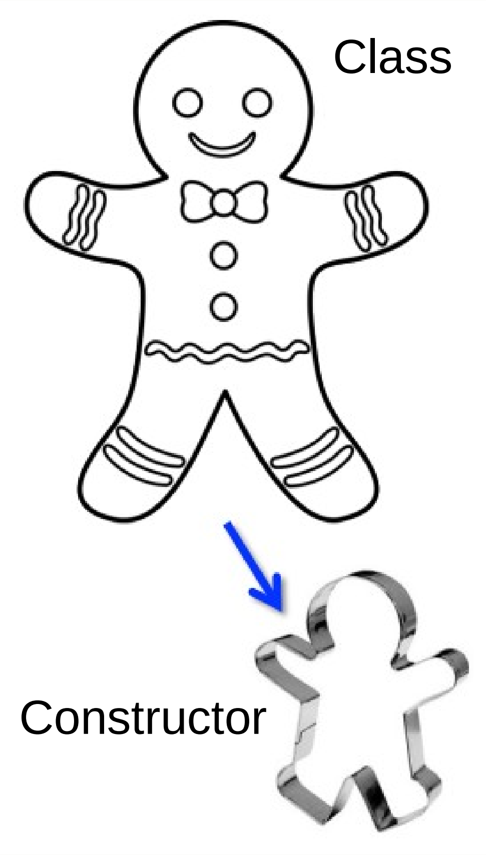
How to construct an object

- A **class is a recipe to build an object**
- Structure of attributes is defined in class
- Attributes are filled with data when object is *instantiated*, i.e. objects of the same class can have different attribute values



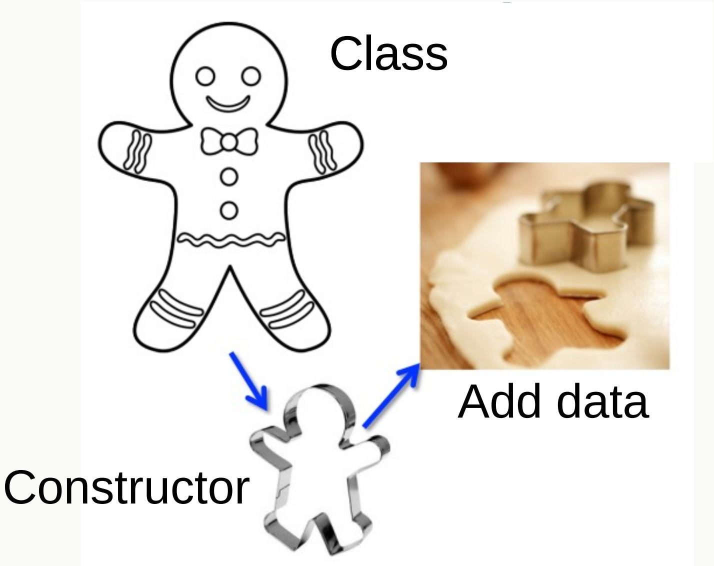
Creating an object: The Constructor

- A constructor is used to set up an object
- The constructor is defined within the class and populates the structure of the object when you first instantiate it



Creating an object: Instantiation

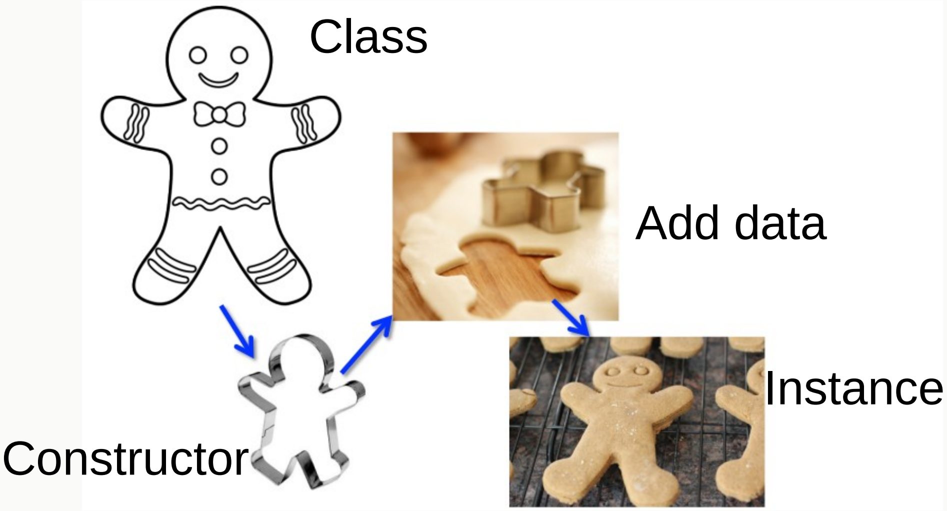
- Creating an object using a constructor from a given class is called *instantiation*
- This creates a new instance of that class in the form of the new object



Creating an object: A New Object

- Once you instantiate a new object using the constructor, you now have an object whose “layout” follows the “blueprint” defined by the class

- The new object contains all the methods defined by that class as well as the default member variables



Creating an object in Python

1) __new__ returns a new instance of your class

2) __init__ populates the instance

Under the hood __new__ is automatically called.
→ Only __init__ required

```python
class Cat:

    # creating a new Cat object
    def __new__(cls, name, age):
        obj = object.__new__(cls)
        return obj

    # initializing the Cat object
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age
```

Creating an object in Python

Positional arguments allow for default values.

```python
class Cat:

    def __init__(self, name: str, age: int = 10):
        self.name = name
        self.age = age

# create new Cat objects
c = Cat("Alice", 5)
c2 = Cat("Bob") # age = 10
```

Excursion: “Dunder” Methods

- __init__(): Constructor to build objects from classes
- __str__(): Print object, similar to __repr__()
- __eq__(): Check for equality
- __len__(): Length of an object

and many more …

```
class Cat:
    def __init__(self, name: str, age: int)  -> None:
        self.name = name
        self.age = age

    def __str__(self) -> str:
        return f"A cat with name {self.name} and age {self.age}"

    def __eq__(self, other: Any) -> bool:
        if not isinstance(other, Cat):
            False
        return all([self.name == other.name, self.age == other.age])

c = Cat("Alice", 5)
c2 = Cat("Bob", 8)

if c == c2: # make use of __eq__
    print("Same cats: ", c) # shortcut for: str(c)
```

Do not call dunder methods directly. Instead, use high-level operations (e.g. str(), + and == operators)

Checking for equality

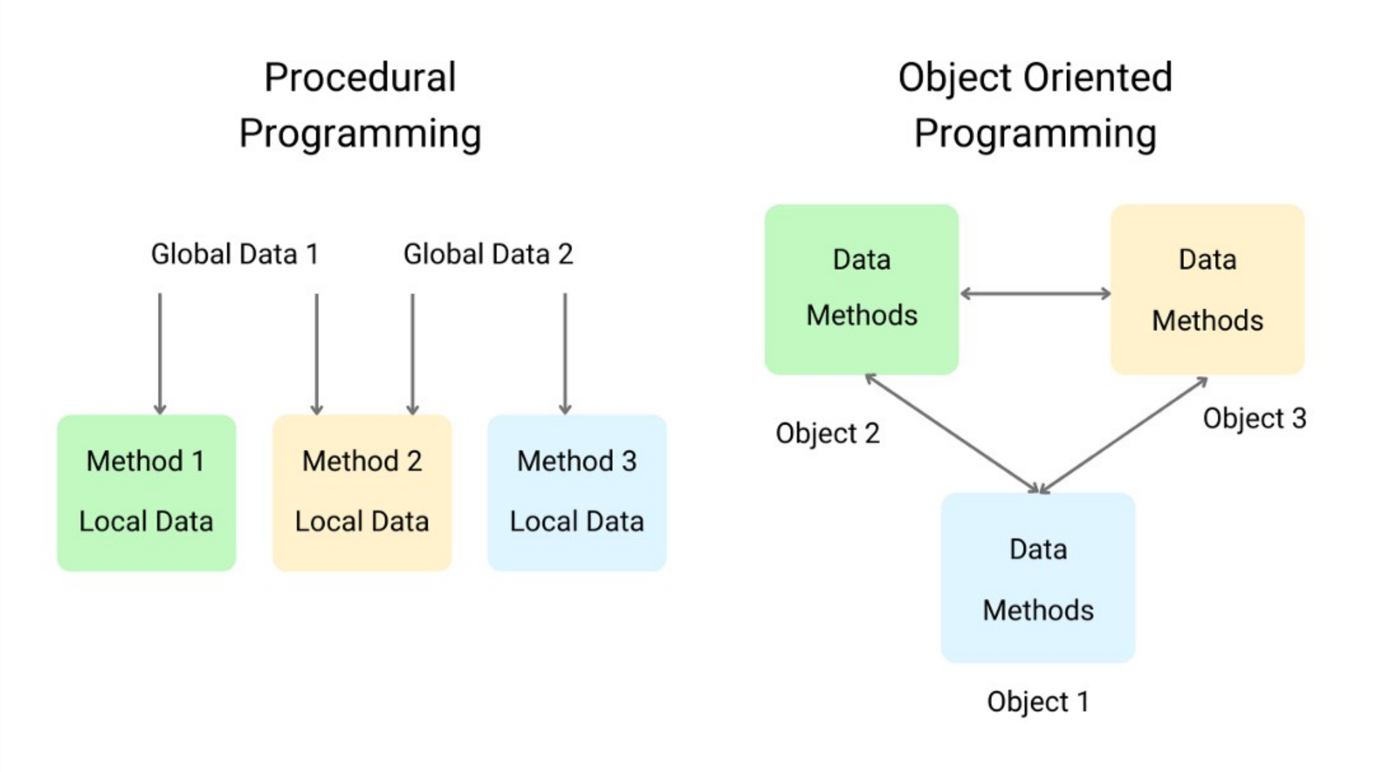
User-defined objects are *mutable*, i.e. retaining identical memory address when changing properties

- Check by pointer reference
- Check by value

Deep copy of object required (deepcopy() library available)

```python
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

c = Cat("Alice", 5)
c2 = Cat("Alice", 5)

# comparison by reference only
assert c != c2

id1, id2 = id(c), id(c2)
assert id1 != id2

# mutable
c.age = 22
assert id1 == id(c)
```

Comparison of Paradigms



How does it look like in Python?

Procedural Programming

```python
# procedural programming

names = ["Alice", "Bob"]
ages = [5, 8]

total_age = sum(ages)
```

NOTE: Both approaches can solve the same problems.

Object-Oriented Programming

```python
# object oriented programming

# class definition
class Cat:

    # constructor
    def __init__(self, name: str, age: int)  -> None:
        self.name = name
        self.age = age


# create a new Cat object
c = Cat("Alice", 5)
c2 = Cat("Bob", 8)

total_age = c.age + c2.age
```

Why are objects used?

Modularize code:

- easier to understand
- easier to maintain
- easier to extend

```python
class Cat:
    def __init__(self, name: str, age: int)  -> None:
        self.name = name
        self.age = age

    # modularise
    def make_sound(self):
        print("Meow.")

# extend class
class Tiger(Cat):

    def make_sound(self):
        print("Groar.")
```

Good News (if you like it so far):

In Python *everything* is an object

Motivation for using OOP

1) **Encapsulation**: limit access to local scope

2) **Inheritance**: extend existing structures

3) **Polymorphism**: reuse existing setups for multi-functionality

4) **Templates**: provide API-like structures for collaborators

1. Encapsulation

1. Encapsulation

User does not need to be concerned with internal workings of the object

User only interacts with interface of object

Makes code much more maintainable — as long the interfaces are stable

```python
class ComplicatedAlgorithm:

    # ...

    def solve():

        # do complicated calculation ...

        return result
```

1. Encapsulation

Closely related to “information-hiding“

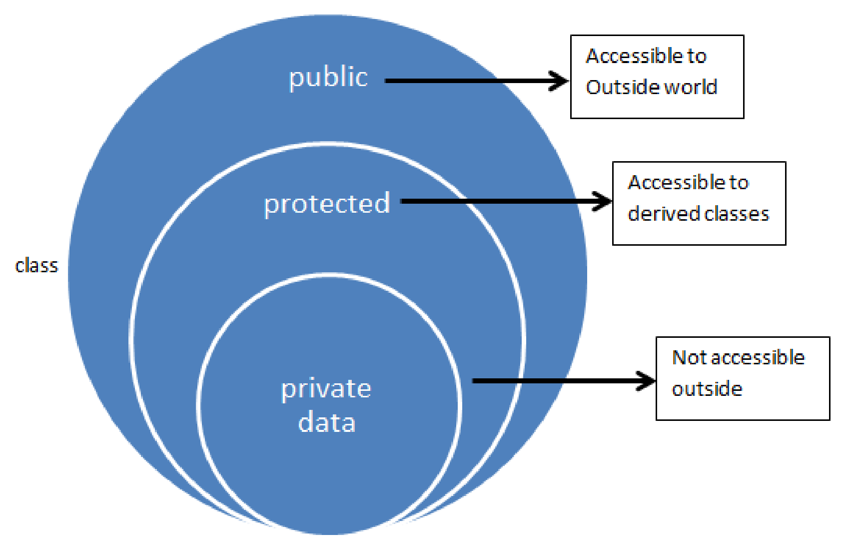
- objects can only be manipulated through controlled methods
- protects user from unintended consequences of directly changing variables

```python
class Cat:

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    def set_age(self, age: int):
        if age < 0: # prevent unphysical input
            raise ValueError
        self.age = age

c = Cat("Alice", 5)
c.set_age(7)
```

How access control usually is done

Prevent direct access to variables by making them private (NOTE: Not for Python)

- Can also have private methods, only usable from within the class
- Other classes/objects can only access this one through its public methods



How access control is done in Python

Not at all …

```python
class Cat:
    super_secret = "password"

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    def set_age(self, age: int):
        if age < 0:
            raise ValueError
        self.age = age

c = Cat("Alice", 5)
c.set_age(3)
c.age = -7 # can still access value
print(c.super_secret) # whoops
```

Alleviation through convention

In Python there is no “private” variable
→ everything is public!

Convention: variables prefixed with an underscore *should* not be accessed (although they could)

ADVANCED: mangling is used for class attributes that one does not want subclasses to use

```
class Cat:

    _please_do_not_access = "secret"

    def _private_method(self):
        print("My thoughts belong to myself only.")
```

```
class Mangling:
    def __mangled_name(self):
            pass
    def normal_name(self):
        pass
t = Mangling()
print([attr for attr in dir(t) if "name" in attr])
# ['_Mangling__mangled_name', 'normal_name']
```

Guide access to attributes

Controlled access to object variables via getter/setter methods

- only* allow retrieval via getter
- only* allow setting via setter

* By force, these can still be circumvented

Use the "@property" decorator

```
class Cat:

    def __init__(self, name: str, age: int):
        self.__name = name
        self.__age = age

    # getter, e.g. print(c.age)
    @property
    def age(self):
        return self.__age

    # setter, e.g. c.age = 7
    @age.setter
    def age(self, value: int):
        if value < 0:
            raise ValueError
        self.__age = value
```

Excursion: Decorator

Decorators are useful to modify the behavior of existing methods.

Used to provide customization options:

- Add/Modify functionality of method without changing its code
- Adding debug/logging information
- Enforcing constraints or permissions on the use of a method

Wrap function with “@<decorator>“

```python
def adapt_default(func):

    def inner():
        print("I got decorated")
        return func(3)
    return inner

@adapt_default # decorator
def ordinary(x = 4):
    return 2 * x

print(ordinary()) # 6
```

Class-wide variables

Sometimes it is useful to have a variable that is shared between all objects of a given class, rather than each instance (object) having its own independent copy.

```python
class Cat:

    # class wide variable
    species_name = "Felis catus"

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age
```

Class-wide methods

Analogously, often useful to have methods that are independent of object instance.

```python
class Cat:

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    @staticmethod
    def general_info():
        print("Cats are nice animals.")

# can be called on the class itself, as well as on any instance of the class
Cat("Alice", 5).general_info()
Cat.general_info()
```

Static Methods – Organizing your code

Static methods are

- independent on state of instance
- related to the class as a whole

Static methods keep code organized, by bundling and encapsulating related functionality within class.

```python
class Cat:

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    @staticmethod
    def general_info():
        print("Cats are nice animals.")

    @staticmethod
    def habitat():
        return "Living rooms all over the world"

c = Cat("Alice", 5)
c.age = 7
c.general_info()
```

Static Methods – Organizing your code

Static methods are used to:

- create utility functions that are logically connected to the class
- create functions that don‘t require access to instance attributes

```
class Cat:

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    @staticmethod
    def general_info():
        print("Cats are nice animals.")

    @staticmethod
    def habitat():
        return "Living rooms all over the world"

c = Cat("Alice", 5)
c.age = 7
c.general_info()
```

Class Methods – Allow implicit access to class

Class methods perform operations on class-level data

- class is implicitly passed as the first argument instead of self

- useful for factory methods: always instantiates the right class, even when subclasses are involved

```python
class Cat:

    # class wide variable
    species_name = "Felis catus"

    @classmethod
    def classmethod(cls):
        # implicitly connected to class
        print(f"Name of this species: {cls.species_name}")

Cat().classmethod()
Cat.classmethod()
```

Classes and their methods

Utilize different method types to ensure correct usage of class.

ADVANCED: Static methods allow to structure code as they are overridable by subclasses.

```python
class Cat:

    # class wide variable
    species_name = "Felis catus"

    def __init__(self, name: str, age: int):
        self.name = name + f" of species {self.species_name}"
        self.age = age

    def method(self):
        # directly connected to object instance
        print(f"Name of this cat: {self.name}")

    @classmethod
    def classmethod(cls):
        # implicitly connected to class
        print(f"Name of this species: {cls.species_name}")

    @staticmethod
    def staticmethod():
        # not related to any other part of class
        print("Cats are nice animals.")

Cat("Alice", 5).method() # Name of this cat: Alice of species Felis catus
Cat.classmethod()        # Name of this species: Felis catus
Cat.staticmethod()       # Cats are nice animals.
```

2. Inheritance

2. Inheritance

Expressing hierarchical relationships between classes:

- Subclass inherits from superclass all methods and attributes (extending)
- Subclass can replace method implementation or data (overriding)

```
class Animal:
    species_name: str

class Cat(Animal):
    species_name = "Felis catus"
    has_claws = True

class Tiger(Cat):
    # overriding
    species_name = "Panthera tigris"

    # extend
    def make_sound(self):
        print("Groar.")

t = Tiger()
print(t.has_claws) # True
```

Complex relationships made easy

Allows to structure complicated code, reuse implementations and specialize the derived class for specific purposes:

- Multiple superclasses allowed (different in e.g. Java)
- Key concept of OOP is to exploit inheritances relationships
- Subclass often called “derived class”, superclass “base class“

```python
class Animal:
    species_name: str

class Pet:
    tamed: True

class Cat(Animal, Pet):
    species_name = "Felis catus"
    has_claws = True

c = Cat()
assert isinstance(c, Animal)
assert isinstance(c, Pet)
```

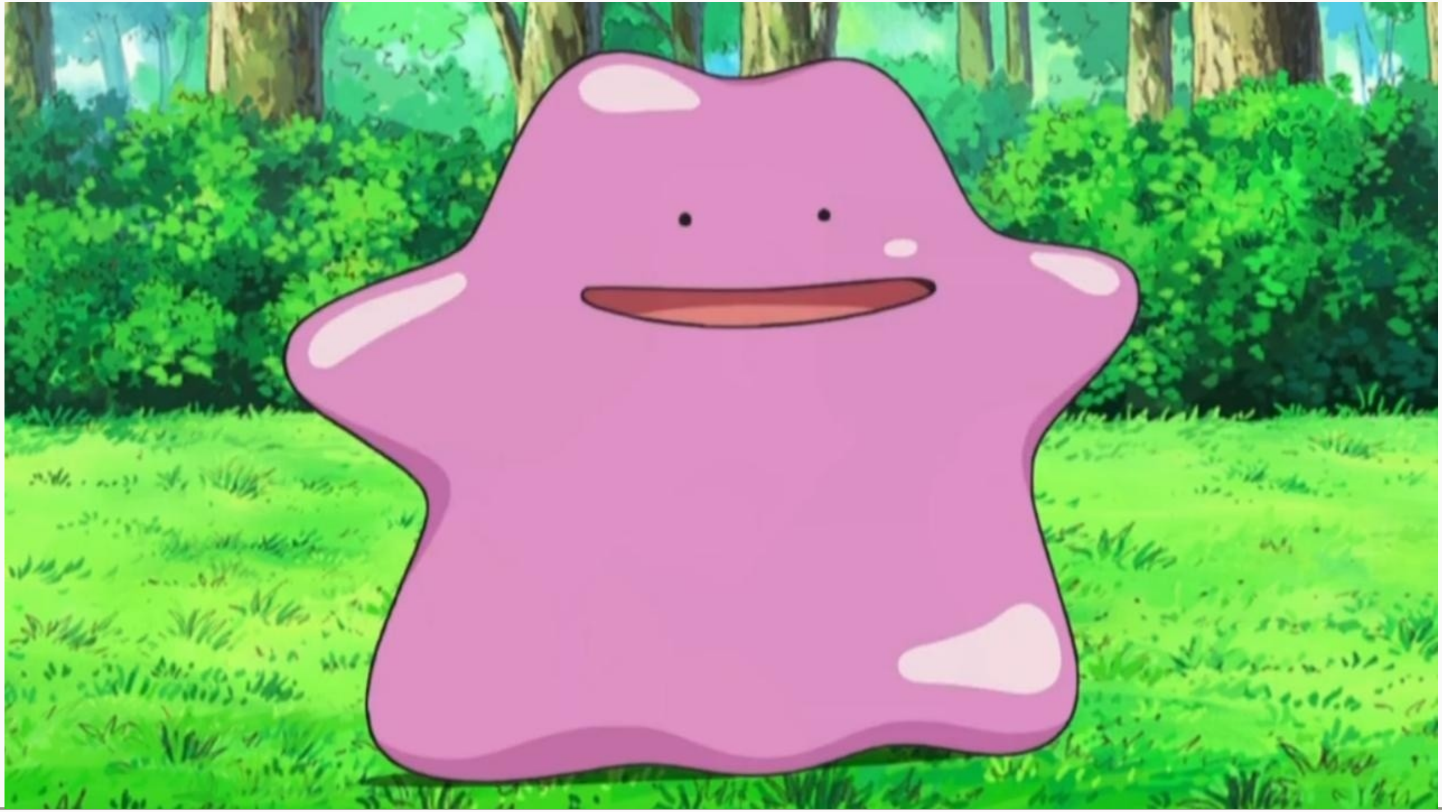
The last super()

The super() method allows to access superclass from subclass:

- Refer to superclass constructor, attributes or methods
- Avoid explicit referral to superclass – especially useful for nested inheritance

```python
class Cat:
    species_name = "Felis catus"

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    def make_sound(self):
        print("Meow.")

class Tiger(Cat):
    species_name = "Panthera tigris"

    def __init__(self, name, age):
        super().__init__(name, age)

    def make_sound(self):
        print("Groar.")

    def encyclopedia(self):
        print(f"The species {self.species_name} is\
            a descendant of {super().species_name}.")
        print(f"While its' ancestors make a sound like:")
        super().make_sound()
        print(f"They sound more like:")
        self.make_sound()

t = Tiger("Shere Khan", 15)
print(t.name, t.age)
t.encyclopedia()
```

3. Polymorphism



3. Polymorphism

Polymorphism allows usage of same interface for different data types, without the need to specify the exact type of the object in advance.

- **Overriding** (dynamic polymorphism): ability of a subclass class to override inherited methods
- **Overloading** (static polymorphism): multiple methods with same name, but different signatures

```python
@overload
def method(input: int) -> None:
    ...
@overload
def method(input: float) -> str:
    ...
def method(input):
    if isinstance(input, int):
        return None
    elif isinstance(input, float):
        return "overloaded"
    raise TypeError

method(234)  # None
method(2.34) # 'overloaded'
```

Inheritance and Polymorphism

"One entity has many forms"

- Two objects may respond in different ways to the same command

Python is a dynamically typed language:

- type checking only at runtime
- type of a variable is allowed to change

```python
class Animal:
    species_name: str

class Cat(Animal):
    def make_sound(self):
        print("Meow.")

class Tiger(Cat):
    def make_sound(self):
        print("Groar.")

class Duck(Animal):
    def make_sound(self):
        print("Quack.")

# Call make_sound on each object in list
for animal in [Cat(), Tiger(), Duck()]:
    animal.make_sound()
```

Duck Typing

In Python, you care about behavior and not about formal type:

> "When it walks like a duck and quacks like a duck, it is a duck."

- Object is considered to exhibit behavior if it implements the necessary attributes or methods, catch all exceptions
- Allows to use any object as long as it provides the expected interface, without the need to explicitly check its type

```python
class Animal:
    species_name: str

class Cat(Animal):
    def make_sound(self):
        print("Meow.")

class Tiger(Cat):
    def make_sound(self):
        print("Groar.")

class Duck(Animal):
    def make_sound(self):
        print("Quack.")

# Call make_sound on each object in list
for animal in [Cat(), Tiger(), Duck()]:
    animal.make_sound()
```

4. Abstract Classes as Templates

4. Abstract Classes as Templates

Define a superclass only as a template for subclasses.

Abstract class (cannot be instantiated):

- contains abstract methods which have a declaration, but no implementation

Concrete class (can be instantiated):

- must define all methods that are left abstract by the superclass

```
class Animal(ABC):
    # abstract class

    @abstractmethod
    def make_sound(self):
        # no implementation
        pass

class Cat(Animal):
    def make_sound(self):
        print("Meow.")

class Duck(Animal):
    def make_sound(self):
        print("Quack.")
```

Common interface facilitates communication in large projects

Abstract classes are useful for defining a common interface for a group of related classes.

- provide blueprint and default implementation for some methods
- ensure that subclasses receive the required functionality and adhere to a consistent interface

```python
# @phd-student: please write a class
# that implements the following.
# Cheers, Your Prof
class Animal(ABC):
    @abstractmethod
    def make_sound(self) -> None:
        """Make a sound."""
        pass

    @abstractmethod
    def has_claws(self) -> bool:
        """Check for claws."""
        pass

    @property
    @abstractmethod
    def weight(self)-> float:
        """Animal weight."""
        pass
```

Abstract implementation enforced

Non-compliance with the abstract class throws an exception at runtime.

```
class Cat(Animal):
    def make_sound(self):
        print("Meow.")

try:
    c = Cat()
except TypeError:
    print("Not all abstract methods instantiated.")
```

ADVANCED: Abstract methods are tracked by name only, i.e. signature is not enforced.

```
class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

class Duck(Animal):
    # tracked by name only
    make_sound = "Quack."
```

Quiz Time

1) Why is OOP more modular than procedural programming?
2) Is there a possibility to make variables private in Python?
3) How can objects be compared for equality?
4) Can multiple decorators be applied to same method?
5) How would you distribute programming work among colleagues?

Tutorial Session (20min)

Please follow the instructions in the Jupyter Notebooks:

```
git clone https://github.com/hoelzerC/Python_OOP.git
```

Further Resources

- https://realpython.com/python3-object-oriented-programming/
- https://github.com/zotroneneis/magical_universe

Required packages in this lecture:

```python
from typing import Callable, overload, Any
from abc import ABC, abstractmethod
```

Part II

Project Setup In Python

THE END