Foundations of Computing

Object Oriented Programming

Ekaterina Vylomova
Summer Term 2023

THE UNIVERSITY OF MELBOURNE

Lecture Agenda

- The OOP Paradigm
- The SOLID Principles
- Object and Class

Programming Paradigms

[The *ART* of Programming]

Programming paradigms set the way you think about a problem, the way you conceptualize it!
The same problem/algorithm can be conceptualized in many different paradigms!

Programming Paradigms

Some popular programming paradigms:

- **Functional**: programs are treated as math functions (R, Lisp; supported by majority of prog. languages. Python: lambda,recursion)
- **Procedural**: step-by-step iterations where common tasks are placed in functions that are called as needed
- **Logic**: follows formal logic. Any programs is a set of sentences in logical form (Prolog)
- **Symbolic**: the program can manipulate its own formulas and program components as if they were plain data (LISP/Prolog)
- **Object-Oriented**: based on the concept of "objects", which can contain data (attributes) and functions (methods).

In Python Everything is an Object!

```
>>> a = 5
>>> type(a)
<class 'int'>
```

Basic data types (defined in C):
`https://github.com/python/cpython/tree/main/Objects`

In Python Everything is an Object!

CSV module: `https://docs.python.org/3/library/csv.html`
Its code:
`https://github.com/python/cpython/blob/3.10/Lib/csv.py`
Recall:

```
reader = csv.DictReader(csvfile)
```

```
class DictReader:
    def __init__(self, f, fieldnames=None, restkey=None, restval=None,
                 dialect="excel", *args, **kwds):
        self._fieldnames = fieldnames   # list of keys for the dict
        self.restkey = restkey          # key to catch long rows
        self.restval = restval          # default value for short rows
        self.reader = reader(f, dialect, *args, **kwds)
        self.dialect = dialect
        self.line_num = 0

    def __iter__(self):
        return self
```

In Python Everything is an Object!

CSV module: `https://docs.python.org/3/library/csv.html`
Its code:
`https://github.com/python/cpython/blob/3.10/Lib/csv.py`
Recall:

```
writer = csv.DictWriter(csvfile, fieldnames =
['first', 'last'])
```

```
class DictWriter:
    def __init__(self, f, fieldnames, restval="", extrasaction="raise",
                 dialect="excel", *args, **kwds):
        self.fieldnames = fieldnames    # list of keys for the dict
        self.restval = restval          # for writing short dicts
        if extrasaction.lower() not in ("raise", "ignore"):
            raise ValueError("extrasaction (%s) must be 'raise' or 'ignore'"
                             % extrasaction)
        self.extrasaction = extrasaction
        self.writer = writer(f, dialect, *args, **kwds)

    def writeheader(self):
        header = dict(zip(self.fieldnames, self.fieldnames))
        return self.writerow(header)
```

The OOP Paradigm I

- Data is structured in the form of *objects*, each of which has a *type* corresponding to a *class*.
- Each problem is conceptualized in terms of *classes*; *class* represents some form of abstraction over objects. A class has methods and properties/attributes.
- Objects *instantiate* a class
- An object *encapsulates* a specific logic of processing of relevant information. Encapsulation ensures security of an object.
- Objects interact with each other via "request – response" interface (via *methods*)

The OOP Paradigm II

- Objects of the same type should process same requests similarly
- Objects can be organized in complex structures, include other objects and be inhereted from one or many objects

The OOP Paradigm III

Three whales of OOP:

- Polymorphism ("multiple forms"): a single interface for entities of different data types
- Inheritance: child classes inherit data and behaviors from parent class
- Encapsulation: containing information in an object, exposing only selected information; hiding the values or state of a structured data object inside a class,

The OOP Paradigm IV

Polymorphism ("multiple forms"): a single interface for entities of different data types

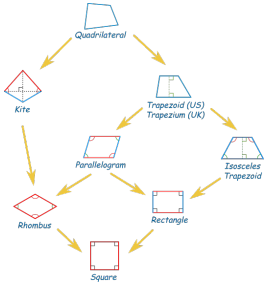
Python: print("Hello") vs. print(1) vs. print(1.0)

The OOP Paradigm V

```
>>> my_file = open('test', 'r')
>>> print(type(my_file))
<class '_io.TextIOWrapper'>
>>> my_file = open('test', 'rb')
>>> print(type(my_file))
<class '_io.BufferedReader'>
```

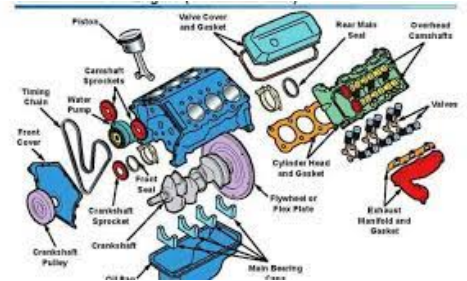
The OOP Paradigm VI

Inheritance: child classes inherit data and behaviors from parent class



The OOP Paradigm VII

Encapsulation: containing information in an object, exposing only selected information. Public (e.g. drive(); check_engine()); Private (engine set up).



SOLID Principles

- **S**ingle responsibility principle: Every class should have only one responsibility.
- **O**pen-closed Principle: Software entities ... should be open for extension, but closed for modification.
- **L**iskov substitution principle (subtyping): An object (such as a class) and a sub-object (such as a class that extends the first class) must be interchangeable without breaking the program
- **I**nterface segregation principle: Many client-specific interfaces are better than one general-purpose interface.
- **D**ependency inversion principle: Depend upon abstractions, [not] concretions.

SOLID Principles

- **S**ingle responsibility principle: Every class should have only one responsibility.

You shouldn't create a single "God object" or "Jack of all trades" object that does everything!

Classes/objects should have specialization!

Need to *decompose* a function/task/problem/concept.

E.W.Dijkstra: "On the role of scientific thought.": "Scientific thought comprises "intelligent thinking" as described above. A scientific discipline emerges with the —usually rather slow!— discovery of which aspects can be meaningfully "studied in isolation for the sake of their own consistency", in other words: with the discovery of useful and helpful concepts. " `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html`

SOLID Principles

- **O**pen-closed Principle: Software entities ... should be open for extension, but closed for modification.

When you create an entity/class that can be re-used, its interface should be designed the way that it stays the same regardless *internal* modifications (i.e. the modifications are compatible)
E.g., you have *CAR* that has *steering wheel, blickers, gas pedal, etc...*
Your new *SUPERCAR* may then extend it with *lamp switcher* but shoudln't remove *steering wheel*.
Public interfaces shouldn't be open for modification.

SOLID Principles

- **L**iskov substitution principle (subtyping): An object (such as a class) and a sub-object (such as a class that extends the first class) must be interchangeable without breaking the program.

Each object has a class (type) but classes are ordered hierarchically.
A child class should provide full functionality of its base class (a user shouldn't notice that the child class is called!).
But don't over-use it! Apply the Occam's razor principle ("entities should not be multiplied beyond necessity")!
Generalize but do not kill yourself in producing too many inherited classes!

SOLID Principles

- **I**nterface segregation principle: Many client-specific interfaces are better than one general-purpose interface.

Swiss knives are evil!
NO single interface for everything!

SOLID Principles

- **D**ependency inversion principle: Depend upon abstractions, [not] concretions.

Introduce abstract classes and inherit from them!
E.g., at driving schools you learn general driving principles and techniques that can be used on any car model/modification.

Objects and Classes

- **Class** is a type that describes the structure of objects.
- **Object** is an instance of a class.

Classes

```
Class C:
  pass
obj_name = class_name() # class constructor
```

```
class Rectangle:
  default_color = "green" # static attribute

  def __init__(self, width, height):
      self.width = width # dynamic attribute
      self.height = height # dynamic attribute
```

"Magic" methods of classes

- __new__(cls,...) – a method called to create a new instance of class *cls*
- __init__(self,) – called after the instance has been created (by __new__())
- __del__– called when the instance is about to be destroyed

See more:
`https://docs.python.org/3/reference/datamodel.html`

“Magic” methods of classes

```
from os.path import join

class FileObject:
'''Wrapper for file objects to make sure the file
       gets closed on deletion.'''
def __init__(self, filepath='~',
                 filename='sample.txt'):
    # open a file filename in filepath in
    # read and write mode
    self.file = open(join(filepath, filename), 'r+')
def __del__(self):
    self.file.close()
    del self.file
```

Instance and class methods

```
class ToyClass:
    def instancemethod(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

Static methods however are primarily used to create utility function and work on data provided to them in arguments.

Abstract classes and methods

Abstract class is a class that contain one or more abstract methods
Abstract method doesn't have an implementation

```
from abc import ABC, abstractmethod

class ChessPiece(ABC):
    # a method that will be used by all child classes
    def draw(self):
        print("Drew a chess piece")

    # an abstract method that should be defined
    # in every child class
    @abstractmethod
    def move(self):
        pass
```

Abstract classes and methods

```
a = ChessPiece() # will raise an error
# TypeError: Can't instantiate abstract class
# ChessPiece with abstract methods move
```

Abstract classes and methods

```
#Queen is a child class of ChessPiece
class Queen(ChessPiece):
    def move(self): #specify ``move''
        print("Moved Queen to e2e4")

# now we can create an instance
q = Queen()
# and call all methods
q.draw()
q.move()
```

Lecture Summary

- The object-oriented paradigm
- The SOLID principles
- Object and class
- Abstract classes and methors