EPFL, CIVIL-127

Programming and software development for engineers

Grading

- Lab 4 is graded
  - due Friday, March 14th, 2025
  - 10% of final grade
- Lab 6
  - due Friday, March 28th, 2025
  - Up to one bonus point, counts towards lab 4
- Mid term
  - Tuesday, April 15th, 2025 at 11am
  - MCQ
  - 50% of final grade
- Final project
  - due Friday, May 23rd, 2025
  - 40% of final grade

Jupyter notebooks

- Web-based, interactive Python environment
- VS Code is a much more powerful environment
  - Code formatters
  - Linters
  - Refactoring tools
  - Built-in debugger
  - A large ecosystem of packages
- Some labs might not work in Jupyter

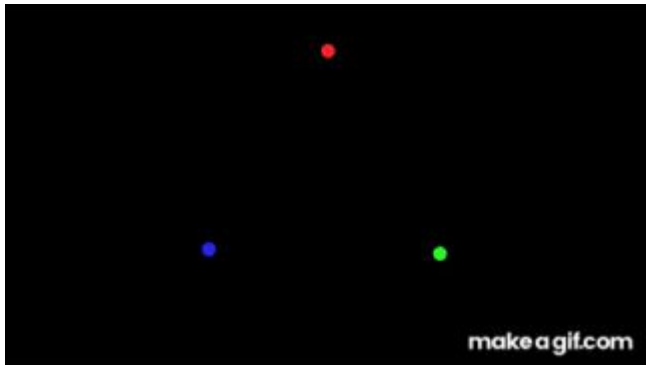
Object Oriented Programming

Object Oriented Programming (OOP) is the most important paradigm to manage software complexity. It has been used in a beneficial way for software of all sizes and all domains, since the 1960s.

OOP: example application

If we want to implement a gravity simulator

- We can model planets and moons as objects
- Each object has many properties (position, velocity, mass, color)
- We can *instantiate* as many copies of an object as we want



What is a Class?

A Class is a blueprint for creating objects. It defines what attributes and methods an object should have

Why use Classes?

- Helps organize code (groups related data (attributes) and code (methods) together)
- Makes code reusable (create multiple objects from the same blueprint)
- Supports Object-Oriented Programming (OOP) principles like encapsulation and inheritance

Defining a class

```
class RunningSum:
    def __init__(self):
        self.s = 0

    def record(self, n):
        self.s += n

    def sum(self):
        return self.s
```

- `class`: keyword, defines the class
- `RunningSum`: pick a name, usually singular, CamelCase, capitalized
- `__init__`: initializer 🪄
  - `self.s` is called an attribute
- `record` and `sum`: `RunningSum`'s methods
- `self`: reference to the instance

Using a class

```python
class RunningSum:
    def __init__(self):
        self.s = 0

    def record(self, n):
        self.s += n

    def sum(self):
        return self.s
```

```python
blue_cars = RunningSum()
blue_cars.record(10)
blue_cars.record(20)
print(blue_cars.sum())   # prints 30
```

Functions vs methods

```
def f(n):
    ...


class Foo:
    def m(self, x):
        ...
```

- `f` is a function
- `m` is a method, belonging to class `Foo` and takes `self` as the first parameter (*)

Parameters vs arguments

```
def f(n):
    ...


class Foo:
    def m(self, x):
        ...


f(10)
Foo().m(20)
```

- `n`, `self`, and `x` are parameters
- `10`, `Foo()`, and `20` are arguments
- Some people/programming languages use the terms formal and actual parameters

camel vs snake vs Pascal vs Kebab case

- Snake case
  - fancy_thing
  - Use it for naming local variables and parameters
  - Use it for filenames
  - Use it for function and method names
- Camel case
  - fancyThing
  - the capital letters make humps, like a camel
- Pascal case
  - FancyThing
  - Camel case with first letter also capitalized
  - Use it for class names
  - (some people call this camel case or StudlyCaps)
- Kebab case
  - fancy-thing
  - Some people use it for filenames

When in doubt, check PEP 8 or see what other pieces of code are doing

Class == Namespace + dict

```python
def running_sum_init():
    return {"s": 0}

def running_sum_record(running_sum, n):
    running_sum["s"] += n

def running_sum_sum(running_sum):
    return running_sum["s"]

blue_cars= running_sum_init()
running_sum_record(blue_cars, 5)
print(running_sum_sum(blue_cars))  # prints 5
```

```python
class RunningSum:
    def __init__(self):
        self.s = 0

    def record(self, n):
        self.s += n

    def sum(self):
        return self.s

blue_cars = RunningSum()
blue_cars.record(5)
print(blue_cars.sum())  # prints 5
```

You can think of a class as a namespace with a dictionary to store attributes

Benefits

Object Oriented Programming provides several benefits, such as:

- Data encapsulation
- Multiple instantiation
- Modularity

Benefits: Data Encapsulation

```
class RunningSum:
    def __init__(self):
        self.s = 0

    def record(self, n):
        self.s += n

    def sum(self):
        return self.s
```

You can visually inspect that `self` is keeping track of the running sum by only looking at the code in `RunningSum` (*)

Benefits: Multiple Instantiation

```
class RunningSum:
    def __init__(self):
        self.s = 0

    def record(self, n):
        self.s += n

    def sum(self):
        return self.s
```

```
blue_cars = RunningSum()
red_cars = RunningSum()
blue_cars.record(1)
blue_cars.record(1)
red_cars.record(1)
print(blue_cars.sum(), red_cars.sum())
```

You can create as many instances of a class as you want. Each one gets its own copy of self.

Benefits: Modularity

- Reuse objects across different projects
  - E.g., generic Particle object can be used to simulate planets, fire, water, snow, etc.
  - E.g., generic PlayingCards can be used for different types of card games
- Swap implementations
  - E.g., different Grid implementations

Code layout

Everything in one file, `main.py`

```
class RunningSum:
    def __init__(self):
        self.s = 0

    def record(self, n):
        self.s += n

    def sum(self):
        return self.s

a = RunningSum()
a.record(10)
a.record(20)
print(a.sum())
```

Code layout

Split across multiple files. In file `main.py`:

```
from running_sum import RunningSum

a = RunningSum()
a.record(10)
a.record(20)
print(a.sum())
```

In file running_sum.py:

```
class RunningSum:
    def __init__(self):
        self.s = 0

    def record(self, n):
        self.s += n

    def sum(self):
        return self.s
```

You also need an empty __init__.py file!

Commenting your code

```
class RunningSum:
    """ Tracks the sum of values seen so far. Starting with an initial sum of 0. """

    def __init__(self):
        self.s = 0

    def record(self, n):
        """ Record that n has been seen by including it in the sum. """
        self.s += n

    def sum(self):
        """ Returns the current sum. """
        return self.s
```

Commenting your code

- Focus on class and methods comments
  - That's what people will read first
  - Tools can extract those comments
- Explain why you are doing something
  - The code already explains the how
  - Example:

```
# n isn't divisible by 2; only check odd numbers
for i in range(3, …, 2):
 if n % i == 0:
   …
```

Instances are assigned by reference

```
from running_sum import RunningSum


a = RunningSum()
a.record(123)
b = a
b.record(456)
print(a.sum(), b.sum())   # prints 579, 579
```

- There is only one instance of `RunningSum` in this code snippet
- `a` and `b` both refer to the same instance
- When you mutate `a`, `b` also changes and vice-versa

Instances are assigned by reference

```
x = [1, 2, 3]
y = x
y.append(4)
print(x, y) # prints [1, 2, 3, 4] [1, 2, 3, 4]
```

- It is the exact same behavior as lists, dicts, etc.

Instances are passed by reference

```
class RunningSum:
    def __init__(self):
        self.s = 0

    def record(self, n):
        self.s += n

    def sum(self):
        return self.s

def foo(x):
    x.record(1)

blue_cars = RunningSum()
blue_cars.record(10)
foo(blue_cars)
print(blue_cars.sum())
```

- In this snippet, there is only one instance of `RunningSum`; `blue_cars` and `x` are therefore the same
- `foo` increments the same instance as `blue_cars`
- This is the same behavior as with lists, dicts, etc.

Each instance is unique

```
from running_sum import RunningSum

blue_cars = RunningSum()
red_cars = RunningSum()
print(blue_cars == red_cars)  # prints False
```

- Each instance is unique
- `==` between different instances returns False
- Need to be careful if you want specific behavior with dicts/sets or want to sort lists

dunder methods

```
class RunningSum:
    def __init__(self):
        self.s = 0

    def record(self, n):
        self.s += n

    def sum(self):
        return self.s

    def __eq__(self, other):
        return self.s == other.s


blue_cars = RunningSum()
red_cars = RunningSum()
print(blue_cars == red_cars)  # prints True
```

- Short for "*d*ouble *under*score methods"
- Also called "magic" methods
- Customize the behavior of equality, type conversions, etc.
- Nearly 100 different methods!

Common dunder methods

Method	Purpose
`__init__()`	Constructor (initialize an object)
`__str__()`	String representation (`print(obj)`)
`__repr__()`	Official string representation (`repr(obj)`)
`__len__()`	Define behavior of `len(obj)`
`__getitem__()`	Access items like `obj[index]`
`__setitem__()`	Set item value `obj[index] = value`
`__delitem__()`	Delete item with `del obj[index]`
`__eq__()`, `__lt__()`, `__gt__()`	Compare objects (`==`, `<`, `>`)
`__add__()`, `__sub__()`	Define behavior for `+` and `-`

private attributes and methods

```
class RunningSum:
    def __init__(self):
        self.s = 0

    def record(self, n):
        self.s += n

    def sum(self):
        return self.s


blue_cars = RunningSum()
blue_cars.s = 100
blue_cars.record(1)
print(blue_cars.sum())  # prints 101
```

- By default, attributes and methods can be accessed from outside a class
- If we had named our attribute `sum`, we wouldn't need to define a method to access `self.s`
- In Python, we consider that data encapsulation is not broken because of the "We are all responsible users" philosophy

private attributes and methods

```python
class RunningSum:
    def __init__(self):
        self._s = 0

    def record(self, n):
        self._s += n

    def sum(self):
        return self._s


blue_cars = RunningSum()
blue_cars._s = 100
blue_cars.record(1)
print(blue_cars.sum())  # prints 101
```

- Single underscore is a convention
- Indicates that an attribute or method is meant to be private
    - But does not make it private!

private attributes and methods

```python
class RunningSum:
    def __init__(self):
        self.__s = 0

    def record(self, n):
        self.__s += n

    def sum(self):
        return self.__s


blue_cars = RunningSum()
blue_cars.__s = 100  # creates a new attribute!
blue_cars.record(1)
print(blue_cars.sum())  # prints 1
print(blue_cars.__s)  # prints 100
```

- Double underscores for attributes and methods makes them private (*)

Inheritance

```
class Foo(Bar):

  ...
```

- Inheritance allows a new (child) class to extend behavior from an existing (parent) class
- For example, `Foo` (the child class) inherits `Bar` (the parent class); we can also say `Foo` extends `Bar`
- `Foo` inherits all the attributes and methods from `Bar`

Enum

```
from enum import Enum

class Colors(Enum):
   BLUE_CRAYOLA = "1f75fe"
   BRICK_RED = "cb4154"

a = Colors.BLUE_CRAYOLA
b = Colors.BLUE_CRAYOLA
print(a == b)  # prints True
print(a.value)  # prints 1f75fe
print(Colors("cb4154").name)  # prints BRICK_RED
```

- Enum (Enumeration) is used as a parent class
- Useful when you have a variable that can take one of a limited selection of values (e.g. days of the week, colors, error states, etc.)
- Deriving from Enum improves code readability; values are grouped together

Testing code

Automatically run code to check that it behaves as expected

Unit testing

- Test each component of your software, one unit at a time
- A unit is:
  - Usually a single class
  - Sometimes a group of related classes
- Run your tests whenever you want to check if your code is still behaving correctly

Example Unit Test

```python
import unittest
from running_sum import RunningSum

class TestRunningSum(unittest.TestCase):
    def testRecordAndSum(self):
        s = RunningSum()
        s.record(10)
        s.record(20)
        self.assertEqual(s.sum(), 30)

if __name__ == '__main__':
    unittest.main()
```

- unittest is Python's built-in testing framework used for writing and running unit tests
- Unit tests help verify that individual pieces of code (functions or methods) work correctly
- To use it, import unittest and create test methods by inheriting from unittest.TestCase
- Name your test class "Test" + class name
- Test methods must begin with "test"
- Use `self.assertEqual`, `self.assertTrue`, etc. to validate code

Example Unit Test: it passes

```python
import unittest
from running_sum import RunningSum

class TestRunningSum(unittest.TestCase):
    def testRecordAndSum(self):
        s = RunningSum()
        s.record(10)
        s.record(20)
        self.assertEqual(s.sum(), 30)

if __name__ == '__main__':
    unittest.main()
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Example Unit Test: it fails

```python
import unittest
from running_sum import RunningSum

class TestRunningSum(unittest.TestCase):
    def testRecordAndSum(self):
        s = RunningSum()
        s.record(10)
        s.record(20)
        self.assertEqual(s.sum(), 25)

if __name__ == '__main__':
    unittest.main()
```

```
F
======================================================================
FAIL: testRecordAndSum (__main__.TestRunningSum.testRecordAndSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "[...]test_running_sum.py", line 12, in testRecordAndSum
    self.assertEqual(s.sum(), 25)
    ~~~~~~~~~~~~~~~~~^^^^^^^^^^^^
AssertionError: 30 != 25
----------------------------------------------------------------------
Ran 1 test in 0.001s
FAILED (failures=1)
```

Unit testing caveats

```
import unittest

class RunningSum:
   ...
   def record(self, n):
       self.s += 1
   ...

class TestRunningSum(unittest.TestCase):
   def testRecordAndSum(self):
       s = RunningSum()
       s.record(1)
       s.record(1)
       self.assertEqual(s.sum(), 2)

if __name__ == '__main__':
   unittest.main()
```

- Unit testing does not cover every possible combination
- It is common to miss edge cases
- As you discover bugs, you can add test cases to avoid regressions

Writing testable code

```
from datetime import date, datetime

def print_your_age():
    birth_date_str = input("Enter your birth
date (dd-mm-yyyy): ")
    birth_date =
datetime.strptime(birth_date_str,
"%d-%m-%Y").date()
    today = date.today()
    time_difference = today - birth_date
    print(f"You are {time_difference.days}
days old.")
```

Writing testable code

```
from datetime import date, datetime


def print_your_age():
    birth_date_str = input("Enter your birth
date (dd-mm-yyyy): ")
    birth_date =
datetime.strptime(birth_date_str,
"%d-%m-%Y").date()
    today = date.today()
    time_difference = today - birth_date
    print(f"You are {time_difference.days}
days old.")
```

Difficult to test!

- input/output
  - `input()`
  - `print()`
- Dependency on external state
  - `date.today()`

Step 1: refactor

```
from datetime import date, datetime

def your_age(user_input, today):
   birth_date = datetime.strptime(user_input,
"%d-%m-%Y").date()

   time_difference = today - birth_date
   return f"You are {time_difference.days} days
old."

def print_your_age():
   birth_date_str = input("Enter your birth date
(dd-mm-yyyy): ")
   r = your_age(birth_date_str, date.today())
   print(r)
```

- `your_age` is going to be testable
  - We remove the I/O
  - We inject our dependency (`today`)
- `print_your_age` can remain untested

Step 2: implement tests

```
from datetime import datetime
import unittest
from your_age import your_age

class TestYourAge(unittest.TestCase):
    def test(self):
        today = datetime.strptime("26-02-2025", "%d-%m-%Y").date()
        r = your_age("01-01-2025", today)
        self.assertEqual("You are 56 days old.", r)

if __name__ == '__main__':
    unittest.main()
```

Other testing strategies: test doubles

- Test doubles
  - Mocks, spies, stubs, fakes, etc.
- Replace code being tested with a simpler implementation
- ⚠️ you might no longer be testing the actual implementation
- Stubbing out `date.today()` isn't easy

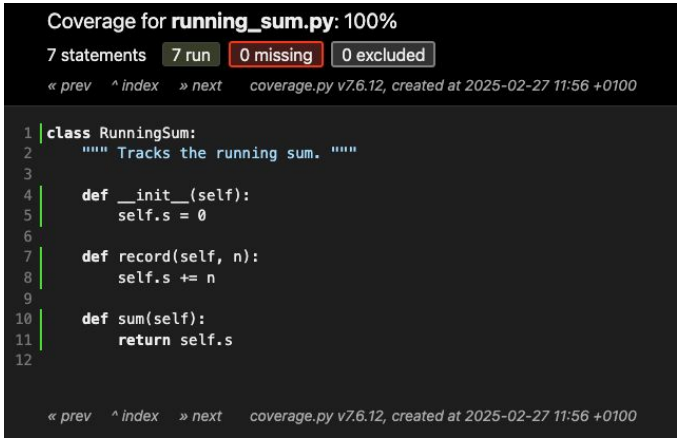
Other testing strategies: test doubles

```
from io import StringIO
import unittest
from datetime import date, datetime
from unittest.mock import Mock, patch
from your_age import print_your_age

class TestYourAge(unittest.TestCase):
   def test(self):
       with patch('builtins.input', return_value="01-01-2025"):
           with patch('sys.stdout', new=StringIO()) as fake_stdout:
               datetime_mock = Mock(wraps=datetime)
               datetime_mock.today.return_value = date(2025, 2, 26)
               with patch('your_age.date', new=datetime_mock):
                   print_your_age()
                   self.assertEqual(fake_stdout.getvalue().rstrip(
                       "\r\n"), "You are 56 days old.")
```

Test coverage

- Test coverage provides information on which lines of code were tested
- `$ pip install coverage`
- `$ coverage run test_running_sum.py`
- `$ coverage html`



Test coverage caveats

```python
def foo(a, b):
    if a or b:
        return True

        ... inside a test ...
        foo(True, False)
```

```python
def foo(a, b):
    if a:
        return True
    if b:
        return True

        ... inside a test ...
        foo(True, False)
```

Code coverage typically measures coverage at the line-level. Two snippets can be functionally equivalent, but have different coverage

Beyond Unit Testing

- A piece of code working in isolation is no guarantee that the code will work once part of a larger system
- This can be addressed by performing integration testing

Design patterns

- Reusable design for many software organization problems
- Helps with figuring out how to name your classes

Model-View-Controller

- Common pattern for UI applications
- Model-View-Controller (MVC) is a way to organize graphical applications
- The Model is responsible for the internal representation of information
- The View is responsible for displaying the model; has access to the model
- The Controller contains the model and view. It proxies input to the model and calls the view.

Model-View-Controller: Sokoban example

```
class SokobanModel:
    def __init__(self):
        self.board = None

    def load(self, level_data):
        ...

    def getPlayerPosition(self):
        ...

    def isEmpty(self, x, y):
        ...

    def isBox(self, x, y):
        ...

    def move(self, dx, dy):
        ...
```

- `SokobanModel` models the playing field and implements player movement rules

Model-View-Controller: Sokoban example

```
class SokobanView:
    def print(self, model):
        ...
```

- `SokobanView` displays the state of the game

Model-View-Controller: Sokoban example

```
class SokobanController:
    def __init__(self):
        self.model = SokobanModel()
        self.view = SokobanView()

    def readFile(self, xsb_file):
        with open(xsb_file, "r") as f:
            self.model.load(f)

    def gameLoop(self):
        while True:
            self.view.print(self.model)
            ...

sokoban = SokobanController()
sokoban.readFile("level1.xsb")
sokoban.gameLoop()
```

- `SokobanController` instantiates the model and view
- `SokobanController` reads the level file
- `SokobanController` runs the game loop:
  - Tell view to display the current state
  - Gets user input
  - Updates model

Model-View-Controller: Sokoban example

```
class SokobanModel:
    def __init__(self):
        self.board = None

    def load(self, level_data):
        ...

    def getPlayerPosition(self):
        ...

    def isEmpty(self, x, y):
        ...

    def isBox(self, x, y):
        ...

    def move(self, dx, dy):
        ...

class SokobanView:
    def print(self, model):
        ...
```

```
class SokobanController:
    def __init__(self):
        self.model = SokobanModel()
        self.view = SokobanView()

    def readFile(self, xsb_file):
        with open(xsb_file, "r") as f:
            self.model.load(f)

    def gameLoop(self):
        while True:
            self.view.print(self.model)
            ...

sokoban = SokobanController()
sokoban.readFile("level1.xsb")
sokoban.gameLoop()
```