# Object Oriented Programming

Jonathan K. Vis

Dept. of Human Genetics, Leiden University Medical Center

j.k.vis@lumc.nl

# Organizing data

- Lists:

```
>>> vectors_x = [0, 1, 2]
>>> vectors_y = [0, 1, 1]
>>> (vectors_x[2], vectors_y[2])
(2, 1)
```

- Nested lists:

```
>>> vectors = [[0, 0], [1, 1], [2, 1]]
>>> vectors[2]
[2, 1]
```

- Dictionaries:

```
>>> vectors = [{'x': 0, 'y': 0},
               {'x': 1, 'y': 1},
               {'x': 2, 'y': 1}]
>>> vectors[2]
{'y': 1, 'x': 2}
```

# Everything is an object

- Python supports many different kinds of data:

```
42    3.14159    'Hello World!'    [1, 1, 2, 3, 5, 8]
{'name': 'Jack', 'age': 25}    True
```

- every object has:
  - a type;
  - an internal data representation (primitive or composite);
  - a set of procedures (functions) for interaction.
- an object is an instance of a type:
  - `42` is an instance of type `int`;
  - `'Hello World!'` is an instance of type `str`.

# Object oriented programming

- create new objects;
- manipulate objects;
- destroy objects:
  - explicitly `del`;
  - or just "forget" about them: Python will destroy inaccessible objects in a process called garbage collection.

Objects are data abstraction:

1. internal data representation using attributes (member variables);
2. an interface (for interaction):
   - procedures (member functions);
   - defines behaviour, but hides implementation.

# Working with objects

1. creating a `class`:
   - define the `class` name;
   - define the `class` attributes (member functions).
2. using the `class`:
   - create new instances of a class;
   - manipulating these instances.

# Define your own types

Use the keyword `class` to define a new type:

- with as parent the `object` type;
- and `Vector` as name.

```
class Vector(object):
    """2d vector class."""

    def __init__(self, x=0, y=0):
        """Initializes a new 2d vector; default: (0, 0)."""
        self.x = x
        self.y = y
```

# More on attributes

Data and procedures (functions) that "belong" to the class:

- Data attributes: the objects that make up the class:
  - a 2d vector is made up of two numbers (`x` and `y`).
- Methods (procedures):
  - functions that *only* work with this class;
  - how to interact with the object;
  - e.g., calculate the length of a vector.

`self` is the current instance of a class.

`def __init__(self, ...)` is a special method to create new instances of a class.

# Creating an instance of a class

```
>>> origin = Vector()
>>> v1 = Vector(2, 1)
>>> print(v1.x, origin.y)
```

- don't provide anything for the `self` argument; Python does that automatically;
- use the dot to access an attribute of an instance;
- `.x` is a member variable.

# Hiding information — separation of concerns

- Sometimes we would like to hide attributes to the outside world, i.e., only usable inside the class.
- in Python we prefix an attribute with _ (underscore) to make it private:

```
class Vector(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
        self._secret = 42     # this is allowed
```

- we agree not to access this attribute directly:

```
>>> v1 = Vector(1, 2)
>>> print(v1._secret)         # this is forbidden
```

# Add a method to the `Vector` class

```
class Vector(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def distance(self, other):
        dx = (self.x - other.x) ** 2
        dy = (self.y - other.y) ** 2
        return (dx + dy) ** .5
```

To use the newly created method:

```
>>> origin.distance(v1)
```

# Print representation of an object

```
>>> v1 = Vector(4, 3)
>>> print(v1)
<__main__.Vector object at 0x7f41ab878450>
```

- per default uninformative;
- define the special method `def __str__(self)`;
- Python calls the `__str__` method automatically when using the `print` function;
- we are in control of what is printed, e.g., for the vector class:

```
>>> print(v1)
<4, 3>
```

# Own print method

The `__str__` function must return a string.

```
class Vector(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def distance(self, other):
        dx = (self.x - other.x) ** 2
        dy = (self.y - other.y) ** 2
        return (dx + dy) ** .5

    def __str__(self):
        return '<' + str(self.x) + ', ' + str(self.y) + '>'
```

# More on types

- get the type of an instance:

```
>>> v1 = Vector(4, 3)
>>> print(type(v1))
<class '__main__.Vector'>
```

- that also works for the class:

```
>>> print(type(Vector))
<type 'type'>
```

- use `isinstance()` to check if an object is a vector:

```
>>> print(isinstance(v1, Vector))
True
```

- what happens here:

```
>>> print(v1.distance(4))
```

# Special operators

- define special operators like: `+`,`-`,`==`,`<`,`>`,`len()`,`...` see: `https://docs.python.org/3/reference/datamodel.html#basic-customization`
- these can be overloaded to work with your class (keep it sensible);
- using the double underscore notation:

```
__add__(self, other)   # self + other
__sub__(self, other)   # self - other
__eq__(self, other)    # self == other
__lt__(self, other)    # self < other
__len__(self)          # len(self)
```

# Example of an overloaded operator

```
class Vector(object):
    ...

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
```

Important: return the right type, e.g., the addition of two vectors results in a vector (not a string)!

```
>>> v1 = Vector(1, -6)
>>> v2 = Vector(3, 4.5)
>>> print(v1 + v2)
<4, -1.5>
```

# The power of OOP

- Bundle together objects that share:
  - common (data) attributes;
  - methods that manipulate these attributes.
- Abstract away implementation by specifying interfaces and behaviour;
- Use inheritance (not covered here) to give an even nicer (layered) abstraction;
- Create own data type on top of what Python provides;
- Reuse code: wrapping code in classes prevents collision of function names;
- Many libraries heavily use classes.

**Questions?**

# Assignment

```
class Fraction(object):
    def __init__(self, numerator, denominator=1):
        self.numerator = numerator
        self.denominator = denominator
```

Use the skeleton to implement a Fraction class containing two integers: `numerator` and `denominator`.

- add, subtract, e.g., $\frac{1}{2} + \frac{2}{3} = \frac{7}{6}$;
- print representation, convert to `float`;
- invert a fraction;
- ...

see: `https://github.com/lumc-python/oop`
or go directly to: `https://classroom.github.com/a/8BnbL9fD`