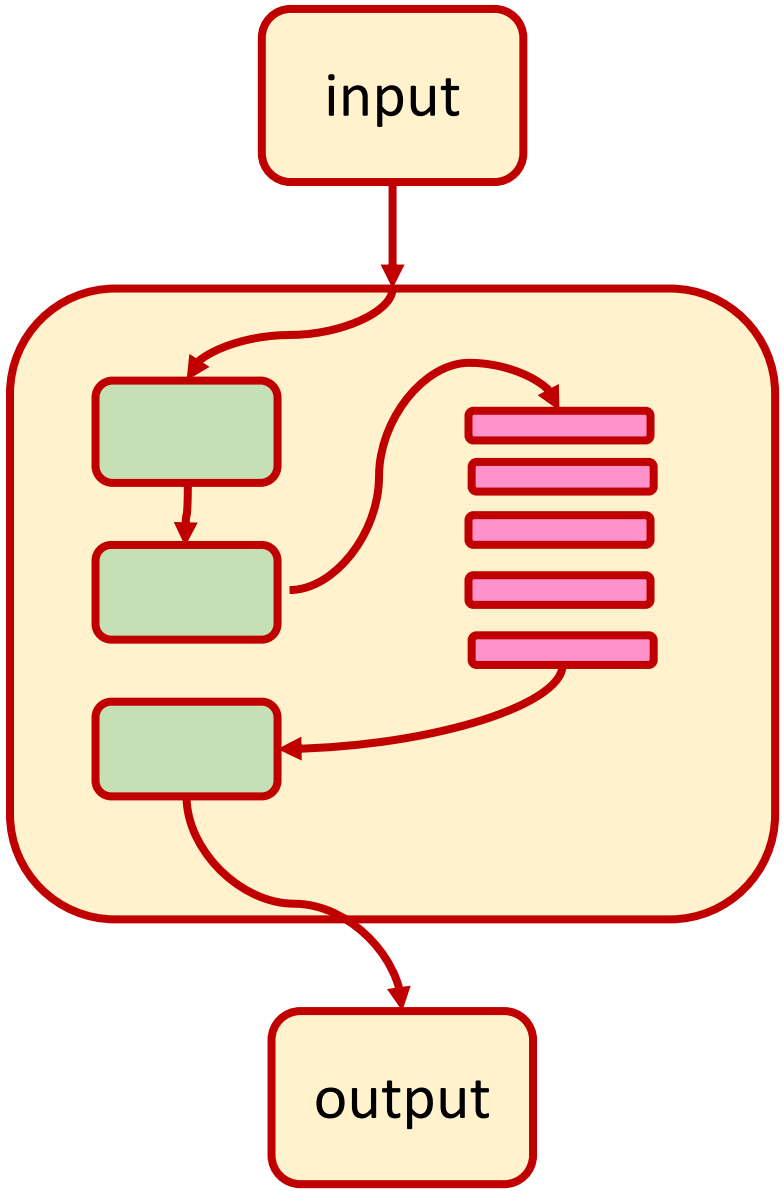
An Introduction to Objects:

Beyond the Procedural Paradigm

The procedural paradigm

Programs are *recipes*: a series of statements that transform our data into visualizations and insight.

In procedural programming, we manage complexity by being **modular**: adhering to top-down programming practices that break down difficult tasks into a sequence of more manageable sub-tasks.



Recipes are a powerful metaphor for data science!

start with raw data

load data

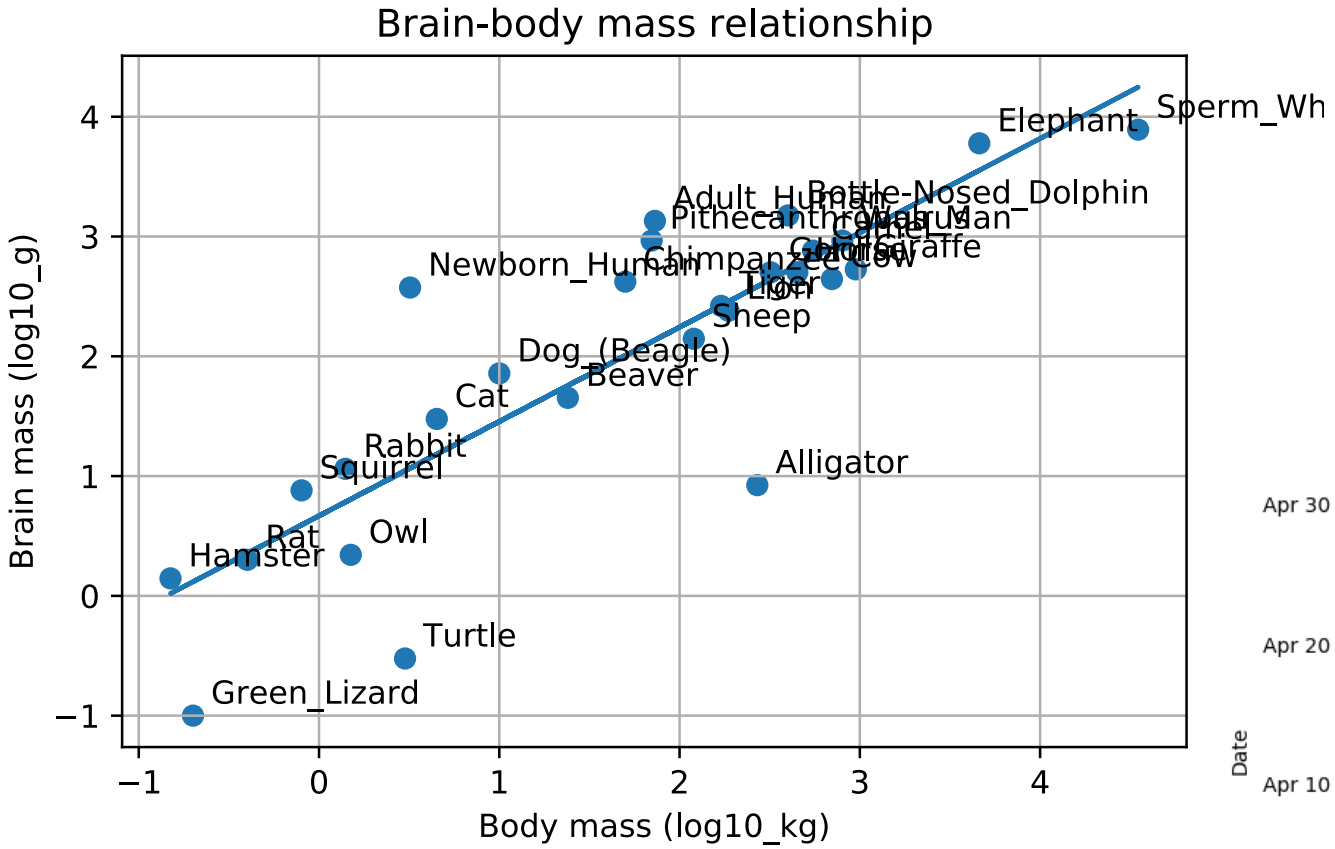
data refinement (munging)

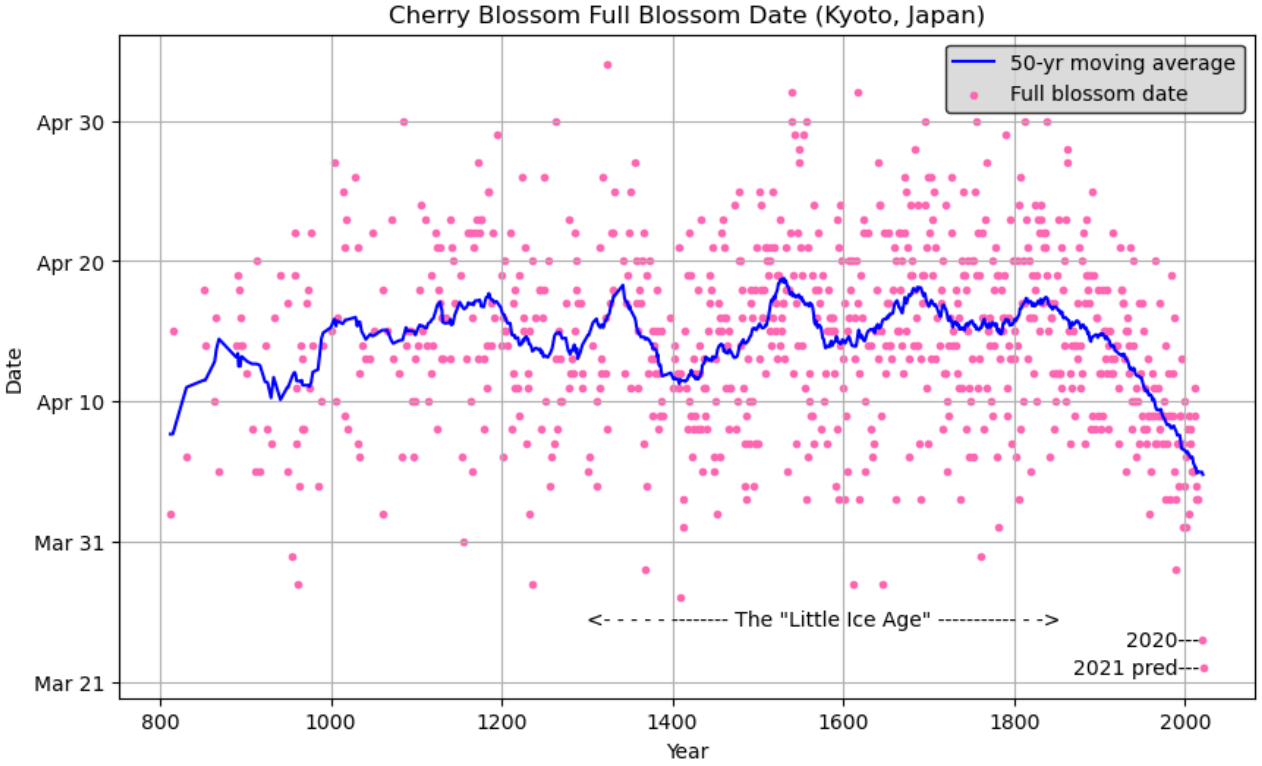
build data structures

processing and visualization

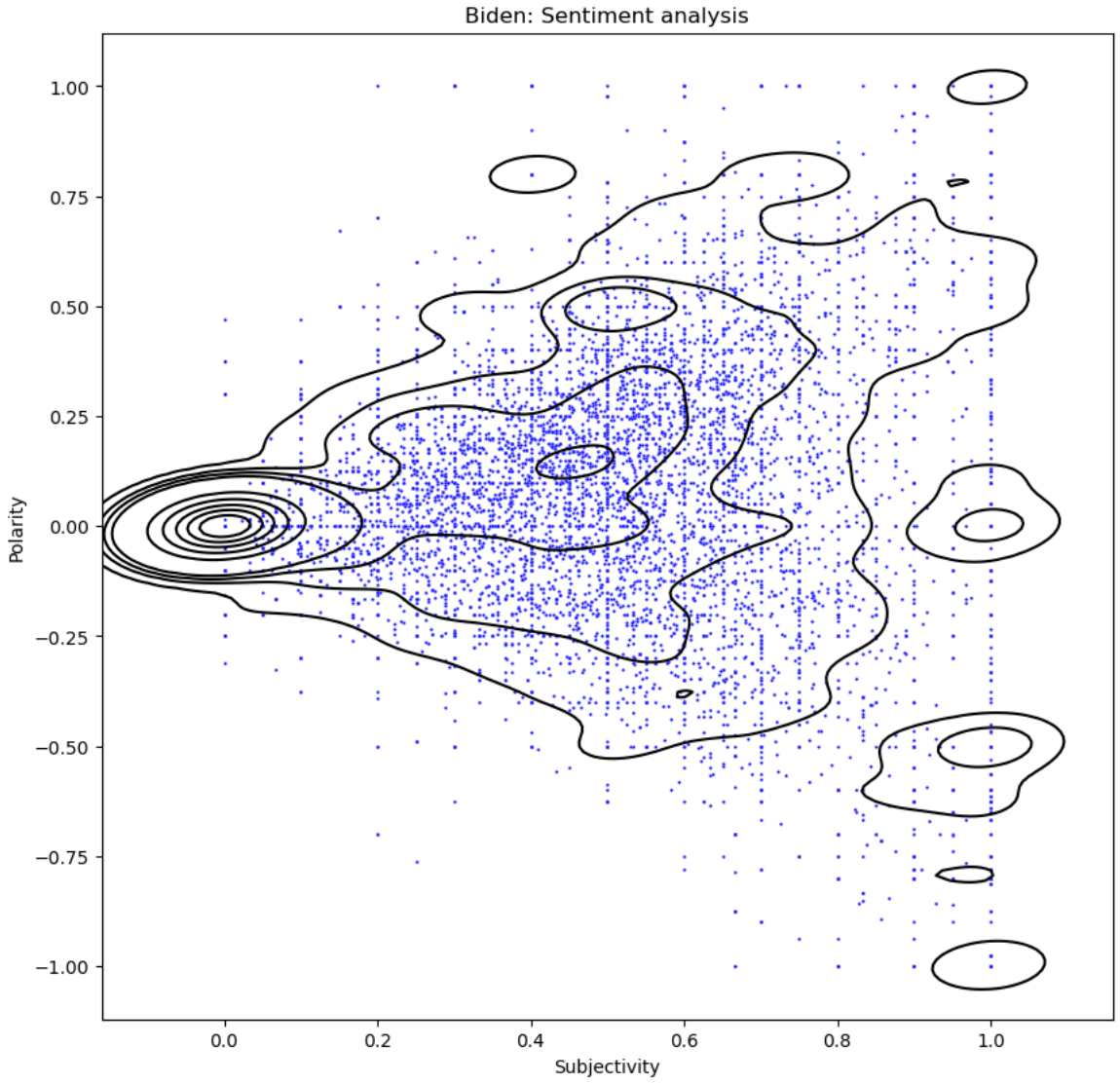
don't forget to label your axes!

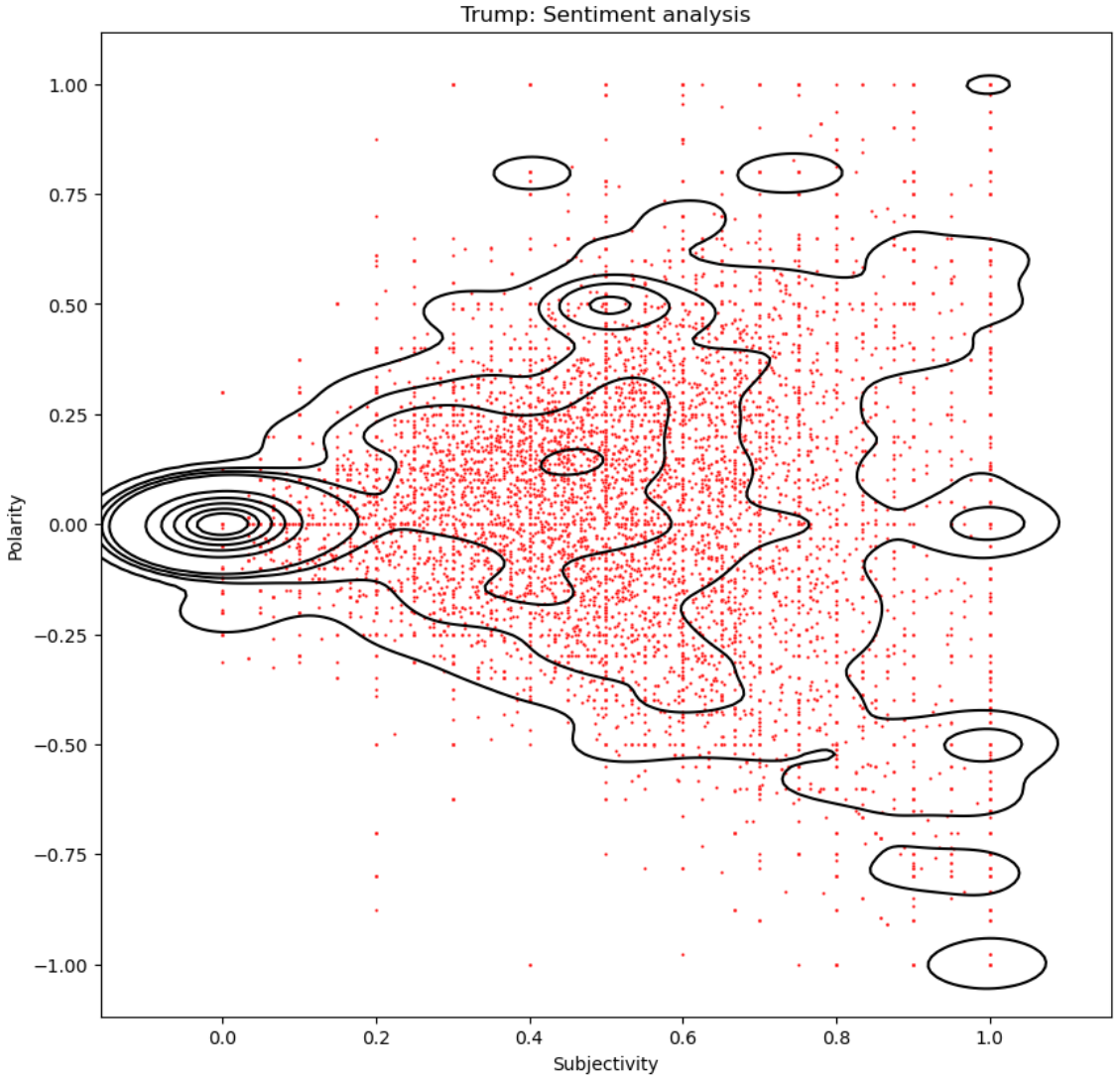
…and it's adequate for a wide range of use-cases!



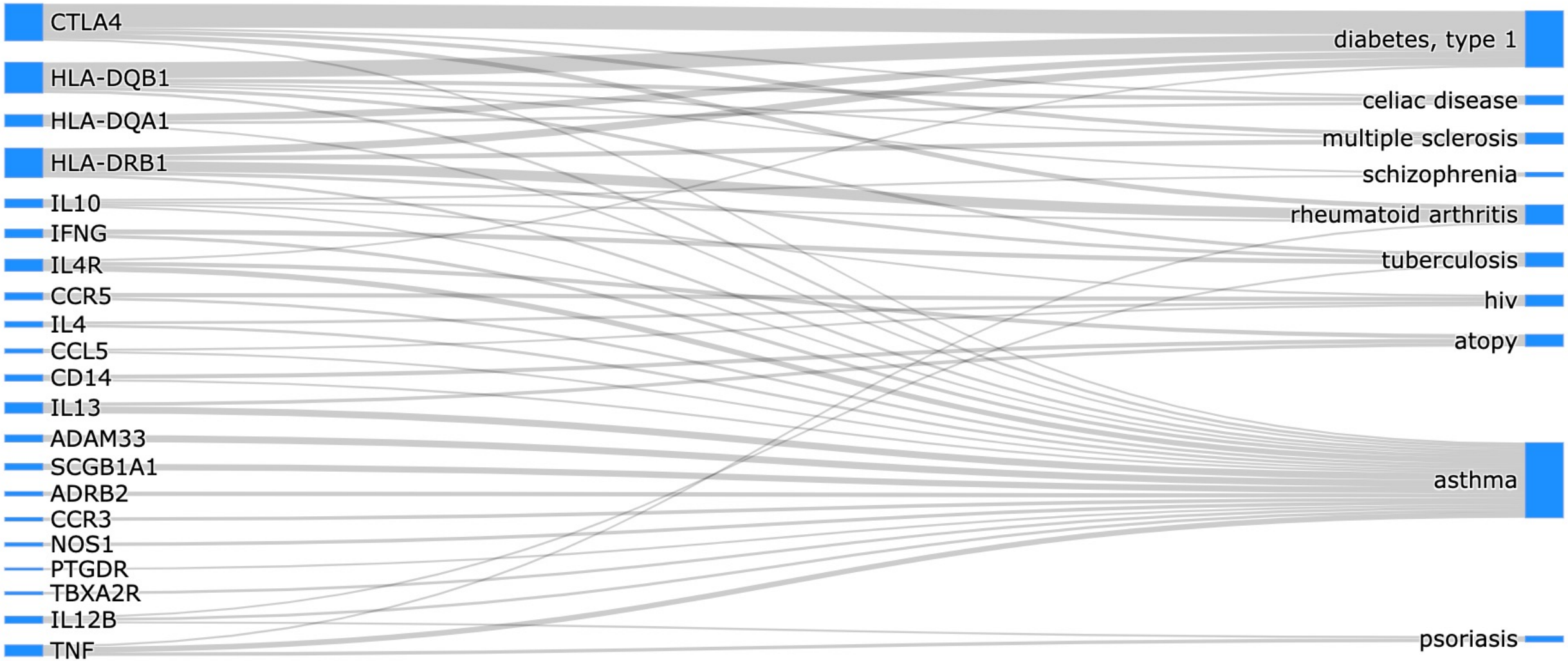


…and it's adequate for a wide range of use-cases!



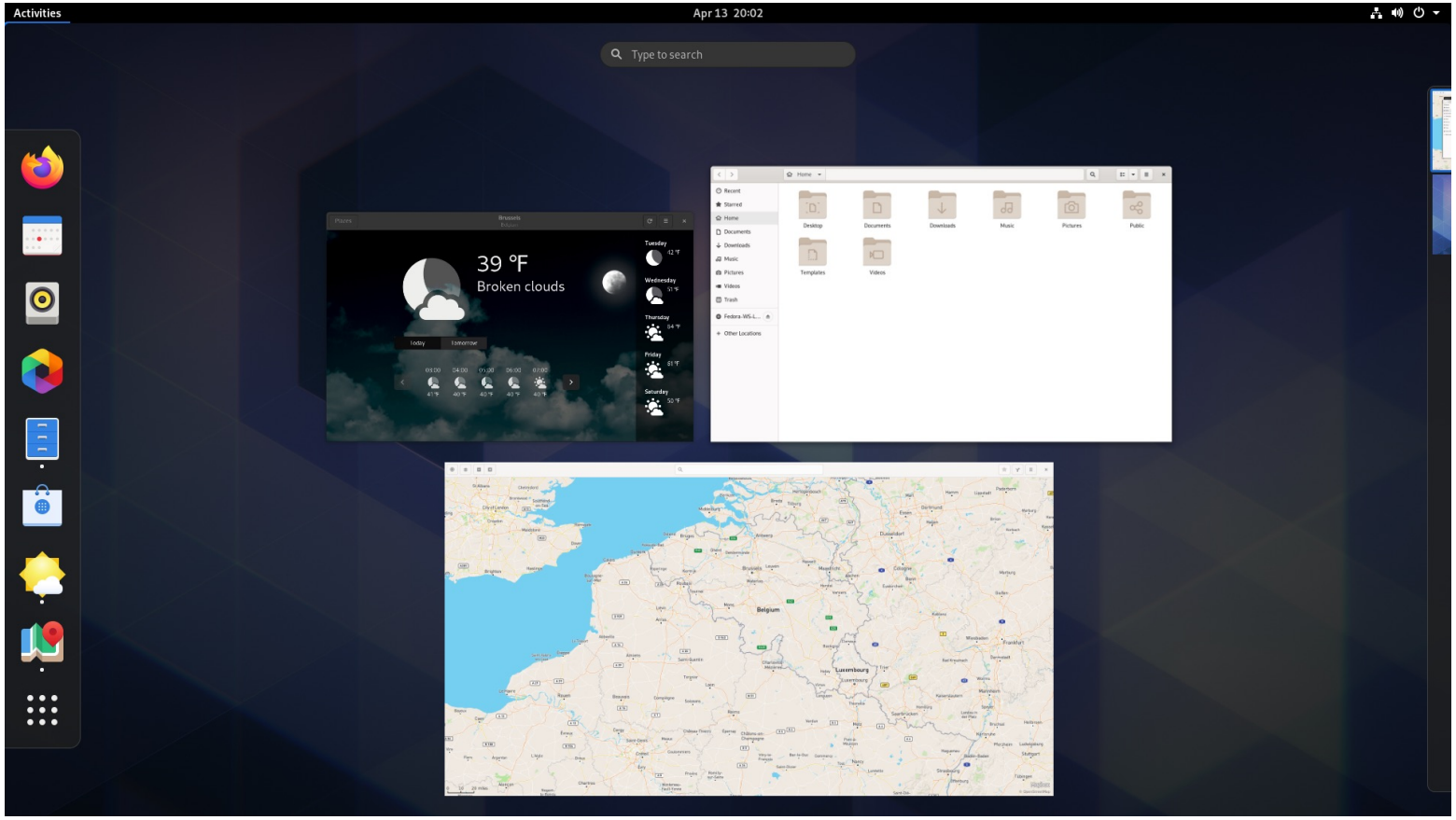


…and it's adequate for a wide range of use-cases!



A plotly sankey diagram revealing the genetic links between **Asthma** and **Type 1 Diabetes**

But consider the Desktop



source: https://www.gnome.org

We **interact** with the desktop **interface** by performing actions on **objects** that each support well-defined **behaviors**.

Objects

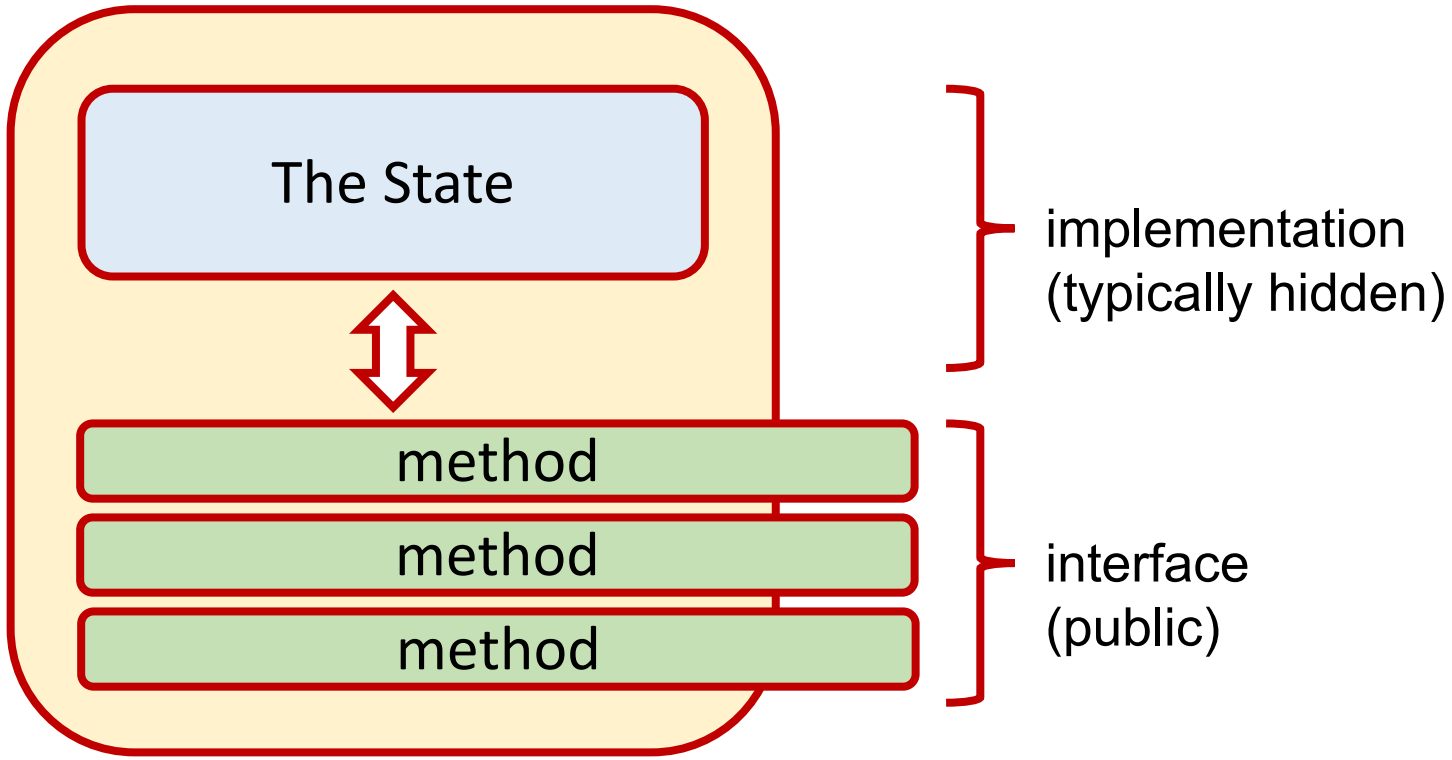
- files and folders
- windows
- apps
- menus
- status bar

Behaviors

- open/close
- move
- resize

The paradigm has clearly changed!

The object paradigm



Objects have fields and attributes that constitute the **state** of the object.

The state is accessed and modified through various **methods** that constitute the object's **interface**.

The layout and organization of the state (i.e., the **implementation**) is usually shielded from the user.

The Interface/Implementation Dichotomy

di·chot·o·my

/dīˈkädəmē/

noun

a division or contrast between two things that are or are represented as being opposed or entirely different.
"a rigid **dichotomy between** science and mysticism"

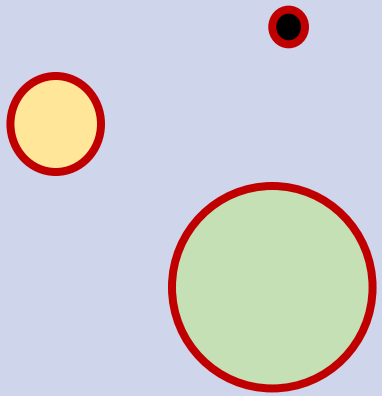
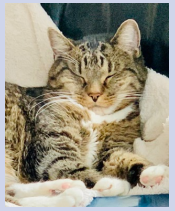
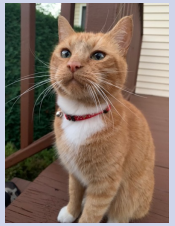
Similar: division | separation | divorce | split | gulf | chasm | difference

We can operate complex machines without knowing what's *under the hood*.

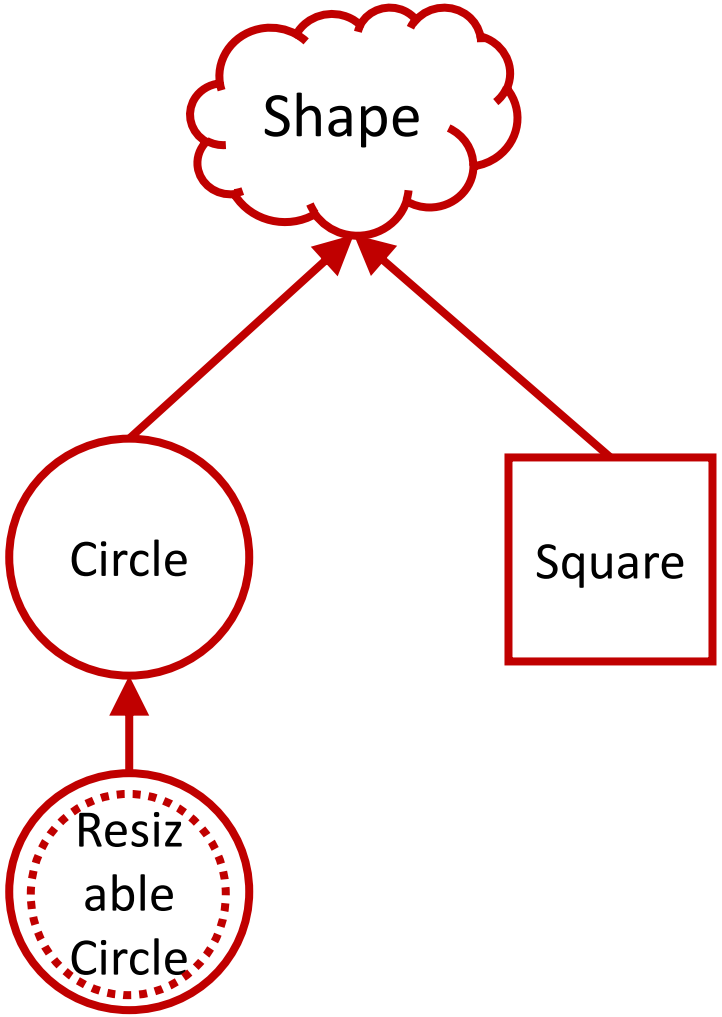
Similarly, to build more complex software, we need to express ideas at a higher level of abstraction with a focus of interface over implementation. Object-oriented thinking enables us to do this.

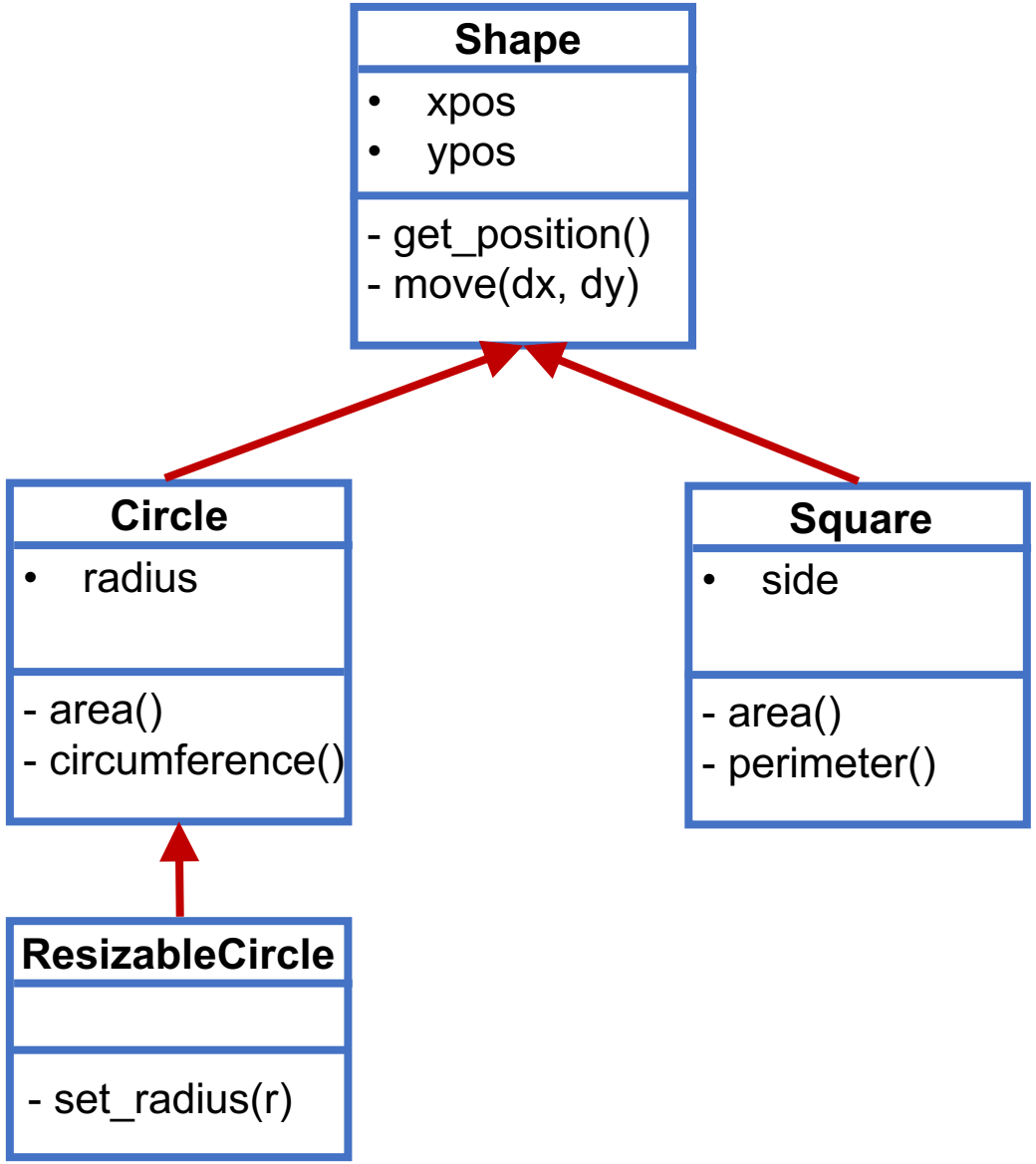
Classes and Objects

Classes define a **type**. It acts as a *template* or blueprint.
We then construct many objects that are **instances** of a particular class.

Class (Type)	List	Circle	Cat	Account
Object Instances	[1, 2, 3] ['Jack', 'Abby'] [] [(0,0), (5,7), (-2,2)]		 my_cat  not_my_cat	MyChecking MySavings . . *etc*.

Class Inheritance





Why objects?

With Objects

```
c = Circle(10)

s = Square(5)

c.area()
314.1592653589793

s.area()
25

c.move(2,5)

c.get_position()
(2, 5)
```

Without Objects

```
def circle_area(a_circle):
    radius = a_circle[1]
    return math.pi * radius ** 2

def square_area(a_square):
    side = a_square[1]
    return side**2

c = [(0,0), 10]     # A circle has a position and a radius
s = [(2,5), 5, 'A'] # A square has a position, side length, and name

circle_area(c)
314.1592653589793

square_area(s)
25



s[2] # Fetch name
'A'

c[2] # Fetch name

Traceback (most recent call last):

  File "<ipython-input-60-5d2164cd07d4>", line 1, in <module>
    c[2] # Fetch name

IndexError: list index out of range
```

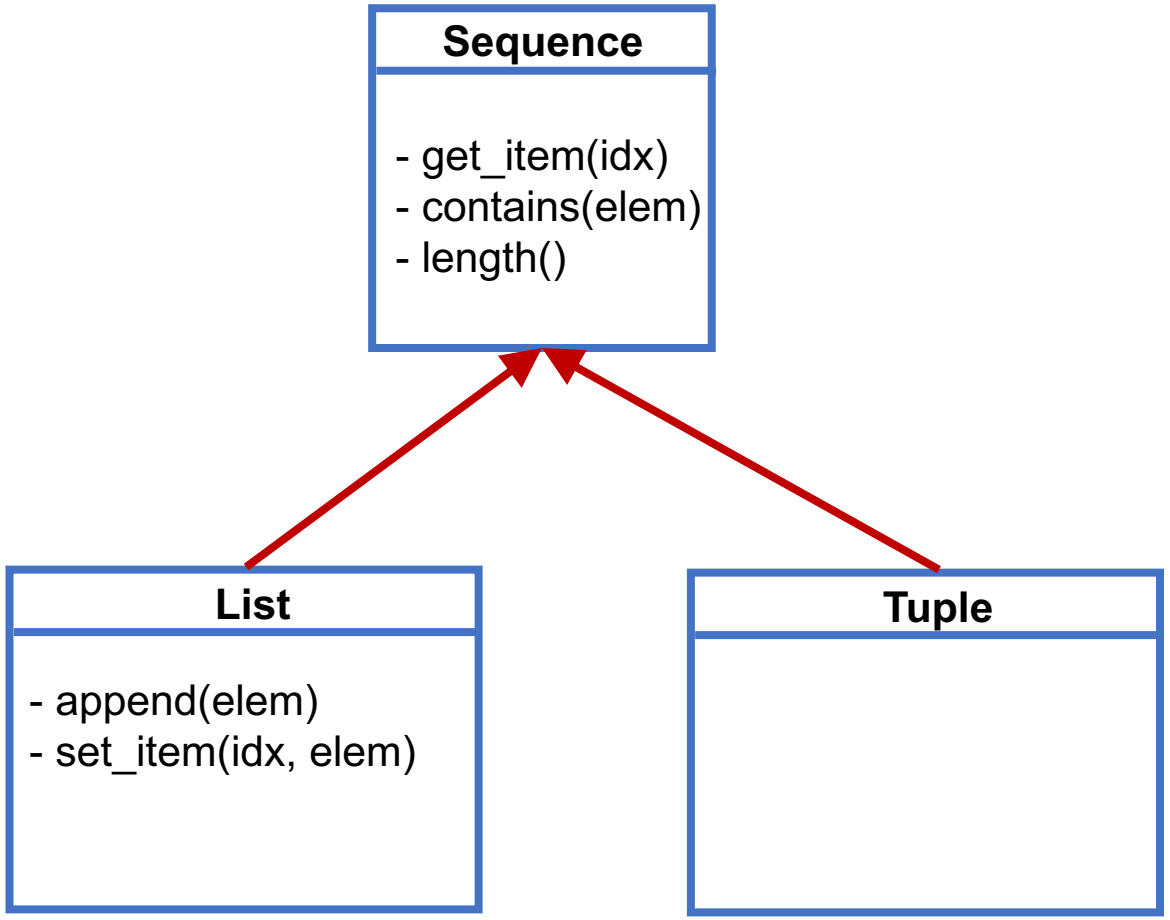
Lists, Tuples, and Dictionaries are objects!

```
L = ['A', 'B', 'C', 'D']

# What is the class/type?

type(L)
list

# Call a method on the object

L.append('E')

L
['A', 'B', 'C', 'D', 'E']

# Pass L to a function - we can still do this!

len(L)
5

L[2]
'C'
```

```
T = ('A', 'B', 'C', 'D')

type(T)
tuple

len(T)
4

T[2]
'C'
```

```
D = {'ann':44, 'reuban':29, 'dachuan':37}

D['reuban']
29

len(D)
3

list(D.keys())
['ann', 'reuban', 'dachuan']
```

How does python implement Lists and Tuples?



Maybe something like this?

No!

Python is implemented in the **C** programming language which does not support classes.

(That came later with C++.)

The Hidden List Implementation of Python 3.9.2

```
static Py_ssize_t
list_length(PyListObject *a)
{
    return Py_SIZE(a);
}

static int
list_contains(PyListObject *a, PyObject *el)
{
    PyObject *item;
    Py_ssize_t i;
    int cmp;

    for (i = 0, cmp = 0 ; cmp == 0 && i < Py_SIZE(a); ++i) {
        item = PyList_GET_ITEM(a, i);
        Py_INCREF(item);
        cmp = PyObject_RichCompareBool(item, el, Py_EQ);
        Py_DECREF(item);
    }
    return cmp;
}
```

-UU-:----F1 **listobject.c** 11% L401 (C/*l Abbrev) ----------

Notice the **for** loop.

To find an element in a list we scan across the items of list one item at a time.

Redundant code in Python 3.9.2

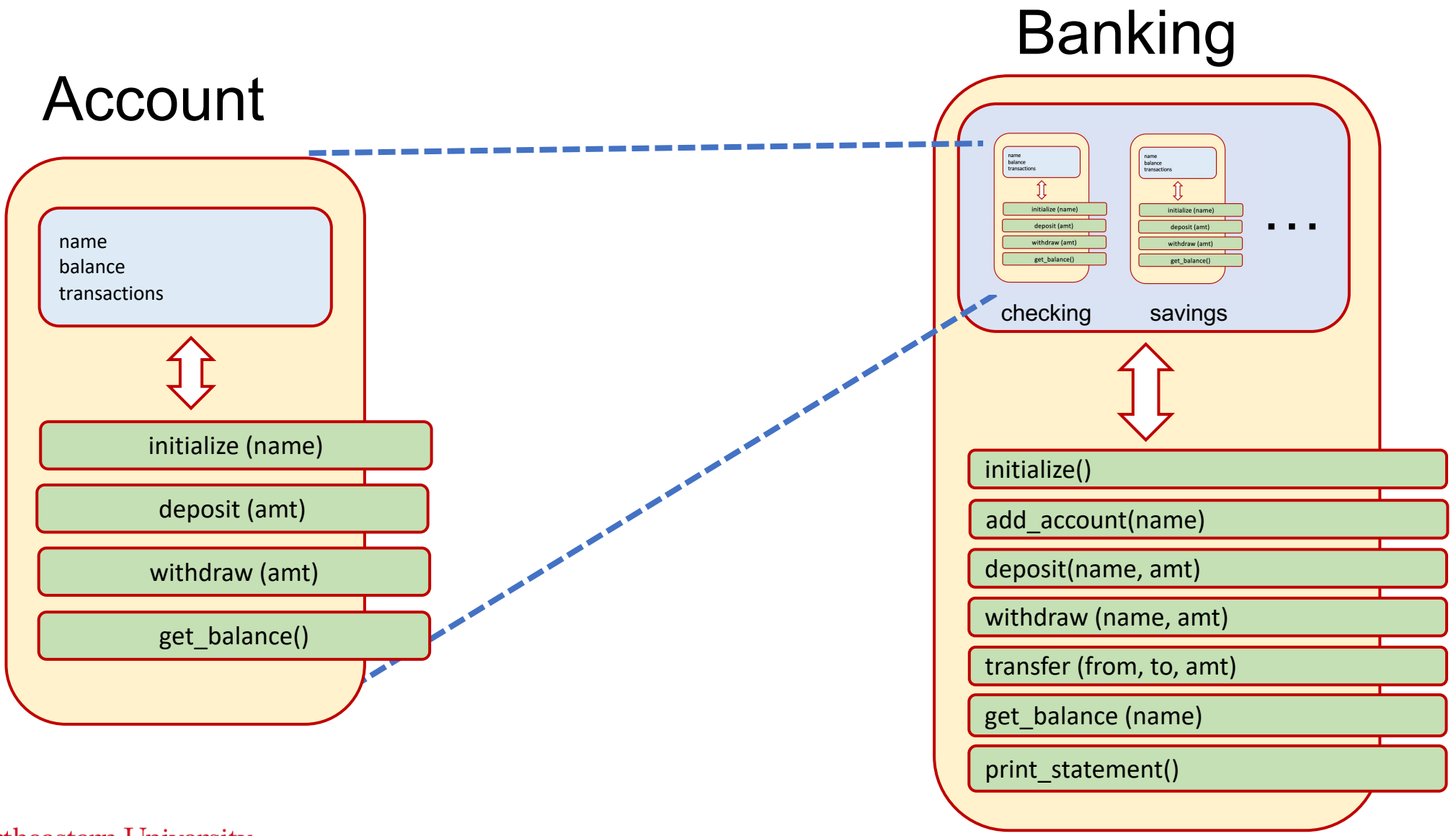
```c
static Py_ssize_t
list_length(PyListObject *a)
{
    return Py_SIZE(a);
}

static int
list_contains(PyListObject *a, PyObject *el)
{
    PyObject *item;
    Py_ssize_t i;
    int cmp;

    for (i = 0, cmp = 0 ; cmp == 0 && i < Py_SIZE(a); ++i) {
        item = PyList_GET_ITEM(a, i);
        Py_INCREF(item);
        cmp = PyObject_RichCompareBool(item, el, Py_EQ);
        Py_DECREF(item);
    }
    return cmp;
}
```

-UU-:----F1 **listobject.c** 11% L401 (C/*l Abbrev) ----------

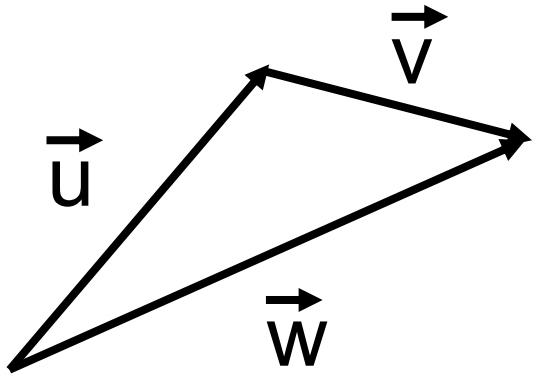
```c
static Py_ssize_t
tuplelength(PyTupleObject *a)
{
    return Py_SIZE(a);
}

static int
tuplecontains(PyTupleObject *a, PyObject *el)
{
    Py_ssize_t i;
    int cmp;

    for (i = 0, cmp = 0 ; cmp == 0 && i < Py_SIZE(a); ++i)
        cmp = PyObject_RichCompareBool(PyTuple_GET_ITEM(a, i), el, Py_EQ);
    return cmp;
}
```

-UU-:----F1 **tupleobject.c** 31% L383 (C/*l Abbrev) ----------

Composition



Operator overloading



This:

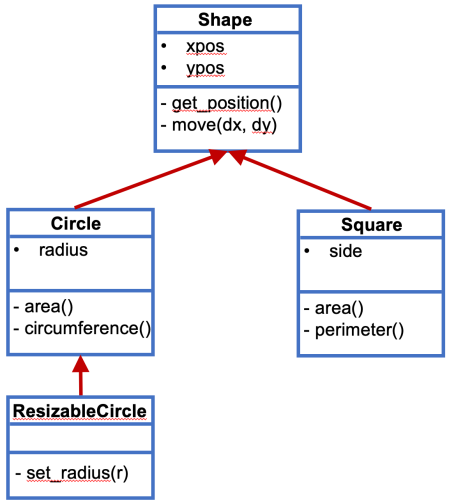
```
u = Vector(3, 4)

v = Vector(2, -1)

w = u + v

w
5i + 3j

u * v  # dot product
2
```

Not this:

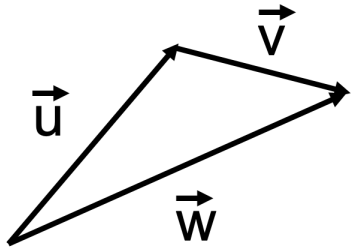
```
u = [3, 4]

v = [2, -1]

w = add(u, v)

w
[5,3]

dot_product(u, v)
2
```

The advantages of object-oriented design

Code re-use
(inheritance)

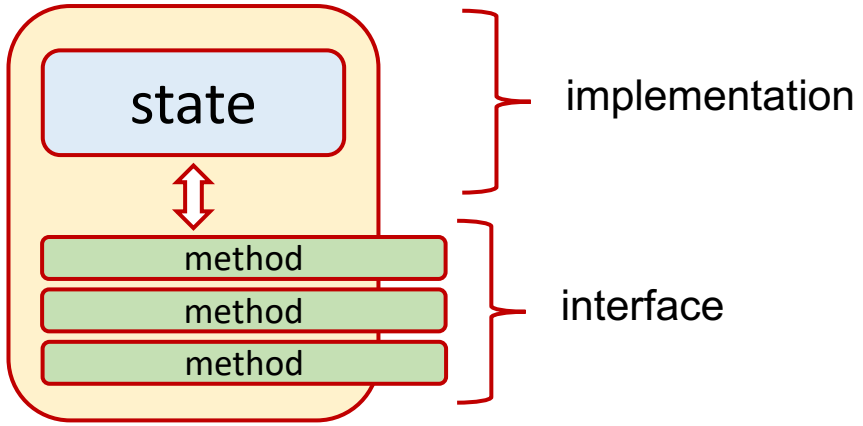


Expressive syntax
(operator overloading)



$$w = u + v$$

Managing complexity
(separation of implementation
and interface)



Object-Oriented Python

Let’s construct some objects!

(Python syntax for creating classes and objects)