Abstract Data Types (I)
(and Object-Oriented Programming)

Jordi Cortadella and Jordi Petit
Department of Computer Science

How many horses can you distinguish?

Two examples

# Main loop of binary search

while left <= right:
    i = (left + right)/2
    if x < A[i]: right = i-1
    elif x > A[i]: left = i+1
    else: return i

# Main loop of insertion sort

for i in range(1, len(A)):
    x = A[i]
    j = i
    while j > 0 and A[j-1] > x:
        j = j-1
    A[j] = x

Variables used (5):
A, x, left, right, i
(only 3 modified)

Variables used (4):
A, x, i, j
Hiding details: abstractions

Different types of abstractions

Concept maps are hierarchical: why?

The computer systems stack

Each level has few items

Image Credit: Christopher Batten, Cornell University
Our challenge

- We need to design large systems and reason about complex algorithms.
- Our working memory can only manipulate 4 things at once.
- We need to interact with computers using programming languages.
- Solution: abstraction
  - Abstract reasoning.
  - Programming languages that support abstraction.
- We already use a certain level of abstraction: functions. But it is not sufficient. We need much more.
Data types

- Programming languages have a set of primitive data types (e.g., int, bool, float, str, ...).

- Each data type has a set of associated operations:
  - We can add two integers.
  - We can concatenate two strings.
  - We can divide two floats.
  - But we cannot divide two strings!

- Programmers can add new operations to the primitive data types:
  - gcd(a, b), match(string1, string2), ...

- The programming languages provide primitives to group data items and create structured collections of data:
  - C: array, struct.
  - Python: list, tuple, dictionary.

Abstract Data Types (ADTs)

A set of objects and a set of operations to manipulate them

Data type: Graph

Operations:
- Number of vertices
- Number of edges
- Shortest path
- Connected components

Data type: Polynomial

Operations:
- \( P + Q \)
- \( P \times Q \)
- \( P / Q \)
- \( \text{gcd}(P, Q) \)
- \( P(x) \)
- \( \text{degree}(P) \)

A set of objects and a set of operations to manipulate them:

\[
P(x) = x^3 - 4x^2 + 5
\]

Abstract Data Types (ADTs)

- Separate the notions of specification and implementation:
  - Specification: “what does an operation do?”
  - Implementation: “how is it done?”

- Benefits:
  - Simplicity: code is easier to understand
  - Encapsulation: details are hidden
  - Modularity: an ADT can be changed without modifying the programs that use it
  - Reuse: it can be used by other programs
Abstract Data Types (ADTs)

• An ADT has two parts:
  – **Public** or external: abstract view of the data and operations (methods) that the user can use.
  – **Private** or internal: the actual implementation of the data structures and operations.

• Operations:
  – Creation/Destruction
  – Access
  – Modification

Example: a Point

• A point can be represented by two coordinates \((x, y)\).

• Several operations can be envisioned:
  – Get the \(x\) and \(y\) coordinates.
  – Calculate distance between two points.
  – Calculate polar coordinates.
  – Move the point by \((\Delta x, \Delta y)\).

# Things that we can do with points

\[
\begin{align*}
p1 &= \text{Point}(5.0, -3.2) \quad \# \text{Create a point (a variable)} \\
p2 &= \text{Point}(2.8, 0) \quad \# \text{Create another point}
\end{align*}
\]

\[
\begin{align*}
\text{# We now calculate the distance between } p1 \text{ and } p2 \\
\text{dist12} &= p1.\text{distance}(p2)
\end{align*}
\]

\[
\begin{align*}
\text{# Distance to the origin} \\
r &= p1.\text{distance}()
\end{align*}
\]

\[
\begin{align*}
\text{# Create another point by adding coordinates} \\
p3 &= p1 + p2
\end{align*}
\]

\[
\begin{align*}
\text{# We get the coordinates of the new point} \\
x &= p3.x() \quad \# x = 7.8 \\
y &= p3.y() \quad \# y = -3.2
\end{align*}
\]
OOP is a programming paradigm: a program is a set of objects that interact with each other.

An object has:
- fields (or attributes) that contain data
- functions (or methods) that contain code

Objects (variables) are instances of classes (types). A class is a template for all objects of a certain type.

In OOP, a class is the natural way of implementing an ADT.

Let us design the new type for Point

class Point:
    """A class to represent and operate with two-dimensional points""
    # Declaration of attributes (recommended for type checking)
    _x: float # x coordinate
    _y: float # y coordinate
    def __init__(self, x: float = 0, y: float = 0):
        """Constructor with x and y coordinates""
        self._x, self._y = x, y
    def x(self) -> float:
        """Returns the x coordinate""
        return self._x
    def y(self) -> float:
        """Returns the y coordinate""
        return self._y
    def distance(self, p: 'Point' | None) -> float:
        """Returns the distance to point p
        (or the distance to the origin if p is None)"
        dx, dy = self.x(), self.y()
        if p is not None:
            dx -= p.x()
            dy -= p.y()
        return math.sqrt(dx*dx + dy*dy)
    def angle(self) -> float:
        """Returns the angle of the polar coordinate"
        if self.x() == 0 and self.y() == 0:
            return 0
        return math.atan2(self.y() / self.x())
    def __add__(self, p: 'Point') -> 'Point':
        """Returns a new point by adding the coordinates of two points.
        This is a method associated to the + operator""
        return Point(self.x() + p.x(), self.y() + p.y())
    def __eq__(self, p: 'Point') -> bool:
        """Checks whether two points are equal.
        This is a method associated to the == operator"
        return self.x() == p.x() and self.y() == p.y()
How the class methods are invoked

```python
p1 = Point(5.0, -3.2)  # __init__(5.0, -3.2)
p2 = Point(2.8)        # __init__(2.8, 0)
dist12 = p1.distance(p2)  # distance(p1, p2)
# Distance to the origin
r = p1.distance()       # distance(p1, None)
# Create another point by adding coordinates
p3 = p1 + p2            # Equivalent to p1.__add__(p2)
# We get the coordinates of the new point
x = p3.x()  # x = 7.8
y = p3.y()  # y = -3.2
```

Magic methods

- They are invoked internally to implement certain actions.
- They are not supposed to be invoked by the user.
- Some examples:
  - Arithmetic: `_add__`, `_mul__`, `_div__`, `_truediv__`, `_neg__`, ...
  - Relational: `_eq__`, `_ne__`, `_gt__`, `_ge__`, ...
  - Representation: `_str__`, `_repr__`, ...
  - Class initialization: `_init__`, `_new__`, `_del__`
    - and others

Python naming conventions

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>distance, dot_product, multiply_by_two</td>
</tr>
<tr>
<td>Variable</td>
<td>x, num, num_elements</td>
</tr>
<tr>
<td>Class</td>
<td>Point, CityGraph, ParkingLot</td>
</tr>
<tr>
<td>Public method</td>
<td>distance, get_angle, shortest_path</td>
</tr>
<tr>
<td>Private method</td>
<td>_gcd, _check, _calculate_mean</td>
</tr>
<tr>
<td>Magic method</td>
<td><strong>init</strong>, <strong>add</strong>, <strong>eq</strong>, <strong>str</strong></td>
</tr>
<tr>
<td>Constant</td>
<td>GRAVITY, MIN_DISTANCE, MAX_NUM_PEOPLE</td>
</tr>
<tr>
<td>Module</td>
<td>point.py, city_graph.py, parking_lot.py</td>
</tr>
<tr>
<td>Package</td>
<td>geometry, citygraph</td>
</tr>
</tbody>
</table>

Recommendation:
- use short names for modules and packages
- no underscores for package names

Comment: ThisIsPascalCase, thisIsCamelCase and this_is_under_score_case

Class Point in C++

```c++
// The declaration of the class Point
class Point {
  public:
    // Constructor
    Point(double x, double y);
    // Constructor for (0,0)
    Point();
    // Gets the x coordinate
    double x() const;
    // Gets the y coordinate
    double y() const;
    // Returns the distance to point p
    double distance(const Point& p) const;
    // Returns the distance to the origin
    double distance() const;
    // Returns the angle of the polar coordinate
    double angle() const;
    // Creates a new point by adding the coordinates of two points
    Point operator + (const Point& p) const;
  private:
    double _x, _y;  // Coordinates of the point
};
```
Implementation of the class Point

// The constructor: different implementations
Point::Point(double x, double y) {
    _x = x; _y = y;
}

// or also
Point::Point(double x, double y) : _x(x), _y(y) {}  

They are equivalent, but only one of them should be chosen. We can have different constructors with different signatures.

// The other constructor
Point::Point() : x(0), y(0) {}  

Implementation of the class Point

double Point::x() const {
    return _x;
}
double Point::y() const {
    return _y;
}

double Point::distance(const Point& p) const {
    double dx = x() - p.x();  // Better getX() than x
    double dy = y() - p.y();  
    return sqrt(dx*dx + dy*dy);
}

double Point::distance() const {
    return sqrt(x()*x() + y()*y());
}

Note: compilers are smart. Small functions are expanded inline.

Conclusions

• The human brain has limitations: 4 things at once.
• Modularity and abstraction are for designing large maintainable systems.