Lecture #6: Abstraction and Objects

Pig Contest Rules

• The score for an entry is the sum of win rates against every other entry.
• All strategies must be deterministic functions of the current score! Non-deterministic strategies will be disqualified.
• Winner: 3 points extra credit on Project 1
• Second place: 2 points
• Third place: 1 point
• The real prize: honor and glory
• To enter: submit a file pig.py that contains a function called final_strategy via the command submit proj1-contest by Monday, 2/13.

Decorators: Pythonic Use of Higher-Order Functions

• The syntax
  @expr
def func(expr):
    body
is equivalent to
  def func(expr):
    body
  func = expr(func)
• For example, our ucb module defines decorator trace. After
  from ucb import trace
  @trace
def mysum(x, y):
    return x + y
mysum will print its arguments and return value each time it is called.
• Usually, expr is a simple name, but it can be any expression that
takes and returns a function.

Functional Abstraction

Consider two implementations of polynomial evaluation:

\[
def \text{quadratic_val}(a, b, c, x):
    \text{"The value of } A \times x^2 + B \times x + C."
    \text{return } a \times x^2 + b \times x + c
\]

• Both have the same name, signature, and (for integers) values.
• To use them, that's all we need—the implementations are irrelevant.
• There is a separation of concerns here:
  - The caller (client) is concerned with providing values of x, a, b, and c and using the result, but not how the result is computed.
  - The implementor is concerned with how the result is computed, but not where x, a, b, and c come from or how the value is used.
  - From the client's point of view, \text{quadratic_val} is an abstraction from the set of possible ways to compute this result.
  - We call this particular kind functional abstraction.
• Programming is largely about choosing abstractions that lead to clear, fast, and maintainable programs.

Guidelines for Defining Functions (I)

• Each function should have exactly one, logically coherent and well
  defined job.
  - Intellectual manageability.
  - Ease of testing.
• Functions should be properly documented, either by having names
  (and parameter names) that are unambiguously understandable, or
  by having comments (docstrings in Python) that accurately describe
  them.
  - Should be able to understand code that calls a function without
    reading the body of the function.
• Don't Repeat Yourself (DRY).
  - Simplifies revisions.
  - Isolates problems.

Guidelines for Defining Functions (II)

• Corollary of DRY: Make functions general
  - copy-paste leads to maintenance headaches
• Keep names of functions and parameters meaningful:

<table>
<thead>
<tr>
<th>Instead of</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>turn, is_over</td>
</tr>
<tr>
<td>d</td>
<td>dice</td>
</tr>
<tr>
<td>helper</td>
<td>take, turn</td>
</tr>
</tbody>
</table>

(Bowling example From Kernighan&Plauger): y score
L ball
f frame
Data Abstraction

• Functions are abstractions that represents computations and actions.
• In the old days, one described programs as hierarchies of actions: **procedural decomposition**.
• Starting in the 1970’s, emphasis moved to the data that the functions operate on.
• An **abstract data type** represents some kind of thing and the operations upon it.
• We can usefuly organize our programs around the abstract data types in them.
• We could just organize our documentation into sections describing the abstract data types we conceptually use,
• But modern programming languages tend to have specific features and syntax for this purpose: **object-oriented programming**.

Objects in Python

• In Python 3, every value is an object.
• Varieties of object correspond (roughly) to **classes** (**types**).
• Each object has some set of **attributes**, accessible using dot notation, which are values:
  - `E.Attr`, where `E` is a simple expression and `Attr` is a name, means “the current value of the `Attr` attribute of the value of `E`”
• Among these attributes are those whose values are a kind of function known as a **method**.
• For historical reasons or convenience, there are often alternative ways to access attributes than dot notation:
  - `x.__add__(y)`
  - `x + y`
  - `L.__getitem__(k)`
  - `L[k]`
  - `x.__len__()`
  - `len(x)`
  - `x.__eq__(y)`
  - `x == y`

Floating-point

• It would be nice if we could represent general real arithmetic efficiently, but we can’t.
• Even if we restrict ourselves to the rationals, simple computations can become quite slow (denominators can grow exponentially).
• Since we don’t usually need absolute accuracy, floating-point was devised as a compromise.
• Typically, (i.e., according to the IEEE Floating-point standard, to which Berkeley faculty (Prof. Kahan) made major contributions), the floating-point numbers are the set
  \[
  \{ \pm s \cdot 2^e \mid 0 \leq s < 2^{53}, \quad -1023 \leq e + 53 \leq 1024 \} \cup \{ \pm \infty, -0, \ldots \}
  \]
  allowing us to represent numbers with maximum magnitude up to `2^{1024}` and non-zero magnitudes as small as `2^{-1074}`.
• `s` is the **significand**, `e` is the **exponent**.

Floating-point Approximation Visualized

• To make things manageable, suppose we restrict `s` to the range 0-3, and `e` to the range -3 to 1
• Then the set of positive floating-point numbers would look like this on a number line:

```
0 1 2 3 4 5 6 7 ...
```