Lecture #6: Recursion

Philosophy of Functions (I)

```python
def sqrt(x):
    """Assuming X >= 0, returns approximation to square root of X."""
```

- **Syntactic specification** (signature)
  - Precondition
  - Postcondition

- **Semantic specification**
  - Specifies a **contract** between caller and function implementor.
  - **Syntactic specification** gives syntax for calling (number of arguments).
  - **Semantic specification** tells what it does:
    - **Preconditions** are requirements on the caller.
    - **Postconditions** are promises from the function's implementor.

Philosophy of Functions (II)

- Ideally, the specification (syntactic and semantic) should suffice to use the function (i.e., without seeing the body).
- 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, `sqrt` is an abstraction from the set of possible ways to compute this result.
  - We call this particular kind of abstraction **functional abstraction**.

- Programming is largely about choosing abstractions that lead to clear, fast, and maintainable programs.

Philosophy of Functions (III)

- 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.

Philosophy of Functions (IV)

- Corollary of DRY: Make functions general
  - copy-paste leads to maintenance headaches
- Taking two (nearly) repeated sections of program code and turning them into calls to a common function is an example of **refactoring**.
- 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

Simple Linear Recursions (Review)

- We've been dealing with recursive function (those that call themselves, directly or indirectly) for a while now.
- From Lecture #3:
  ```python
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    if N < 1:
        return 0
    else:
        return N**2 + sum_squares(N - 1)
```
- This is a simple linear recursion, with one recursive call per function instantiation.
- Can imagine a call as an expansion:
  ```python
  sum_squares(3) => 3**2 + sum_squares(2)
  => 3**2 + 2**2 + sum_squares(1)
  => 3**2 + 2**2 + 1**2 + sum_squares(0)
  => 3**2 + 2**2 + 1**2 + 0 => 14
  ```
- Each call in this expansion corresponds to an environment frame, linked to the global frame, as shown here.
Tail Recursion

- We’ve also seen a special kind of linear recursion that is strongly linked to iteration:

```python
def sum_squares(N):
    """The sum of K**2 for 1 <= K <= N."""
    accum = 0
    k = 1
    while k <= N:
        accum += k**2
        k += 1
    return accum
```

- The right version is a tail-recursive function: the recursive call is either the returned value or very last action performed.

- The environment frames correspond to the iterations of the loop on the left, as shown here.

Recursive Thinking

- So far in this lecture, I’ve shown recursive functions by tracing or repeated expansion of their bodies.

- But when you call a function from the Python library, you don’t look at its implementation, just its documentation (“the contract”).

- Recursive thinking is the extension of this same discipline to functions as you are defining them.

- When implementing `sum_squares`, we reason as follows:
  - Base case: We know the answer is 0 if there is nothing to sum (N < 1).
  - Otherwise, we observe that the answer is \(N^2\) plus the sum of the positive integers from 1 to \(N - 1\).
  - But there is a function (`sum_squares`) that can compute \(1 + \ldots + N - 1\) (its comment says so).
  - So when \(N \geq 1\), we should return \(N^2 + \text{sum_squares}(N-1)\).

- This “recursive leap of faith” works as long as we can guarantee we’ll hit the base case.

Recursive Thinking in Mathematics

- To prevent an infinite recursion, must use this technique only when
  - The recursive cases are “smaller” than the input case, and
  - There is a minimum “size” to the data, and
  - All chains of progressively smaller cases reach a minimum in a finite number of steps.

- We say that such “smaller than” relations are well founded.

- We have

  Theorem (Noetherian Induction): Suppose \(\prec\) is a well-founded relation and \(P\) is some property (predicate) such that whenever \(P(y)\) is true for all \(y \prec x\), then \(P(x)\) is also true. Then \(P(x)\) is true for all \(x\).


- More general than the “line of dominos” induction you may have encountered: If true for a base case \(b\), and if true for \(k\) when true for \(k-1\), then true for all \(k \geq b\).

A Problem

```python
def find_first(start, pred):
    """Find the smallest k >= START such that PRED(START)."""
```

Last modified: Tue Feb 2 15:56:07 2016