Introduction to Problem Solving and Programming in Python

http://cis-linux1.temple.edu/~tuf80213/courses/temple/cis1051/
Overview

• Name scope
• Recursion
Scope

• Scope defines the visibility of a name within a block
• Variables declared in a block are only visible inside that block
  • These variables are said to have *local* scope
  • Local variables are also visible inside all sub-blocks of their defined block
• A variable declared in a block does not exist in the top-level namespace
The **global** keyword

- When assigned at the top-level, a variable can be **read** by any defined code block
  - These variables are said to have *global* scope
- In order for a block to modify a top-level variable, the block must declare the variable as **global**
## Using `global`

<table>
<thead>
<tr>
<th>Wrong</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = 0</code></td>
<td><code>x = 0</code></td>
</tr>
</tbody>
</table>
| ```python
def change_val():
    x = 123
```                             | ```python
def change_val():
    global x
    x = 123
```                             |
| `print(x)  # 0`                           | `print(x)  # 0`                           |
| `change_val()`                            | `change_val()`                            |
| `print(x)  # 0`                           | `print(x)  # 123`                         |
What is recursion?

• Recursion is the process of solving a problem by solving smaller instances of the same problem
• The solutions to the sub-problems are combined to give the solution of the parent problem
• This is usually accomplished through use of a function that calls itself
Recursive functions

• A function is recursive when it is defined in terms of itself
  • If function $x()$ eventually calls $x()$

• Each time the function calls itself, it must pass a smaller/simplified version of the argument

• The recursive calls should eventually end (similar to loops)
Loop or recursion?

- Many looping problems can be solved with recursion
- These problems have similar structure
- Many times recursion can offer more elegant solution
Defining a recursive function

• Define a base case
  • The point at which an atomic value is returned without making another recursive call

• Set rules to reduce the data set used to make the recursive call
factorial()

• Purpose
  • Calculate $n!$
  • e.g. $5! = 5 \times 4 \times 3 \times 2$
factorial() recursive solution

```python
def factorial(n):
    if n == 1:
        return n
    return n * factorial(n - 1)
```

Base case

Recursive call

Solving larger problem
With result of smaller problem

Simplified dataset
get_sum()

• Purpose
  • Take a list of numbers and return a sum
  • Operates like Python’s sum() function
get_sum() recursive implementation

```python
def get_sum(items):
    if len(items) == 1:
        return items[0]
    return items[0] + get_sum(items[1:])
```

Diagram:
- Base case: `if len(items) == 1`
- Recursive call: `return items[0] + get_sum(items[1:])`
- Simplified dataset
- Solving larger problem
  - With result of smaller problem
fibonacci()

- Purpose
  - Return the $n^{th}$ numbers from the fibonacci sequence
  - e.g. – the 10th fibonacci number is 55 (1, 1, 2, 3, 5, 8, 13, 21, 34, 55)
def fibonacci(n):
    if n <= 2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)