

Recursion

23 Nov 2010

CMPT140

Dr. Sean Ho

Trinity Western University

What's on for today

■ Recursion

- Call stack
- Application: `factorial()`
- Tail recursion (and why it's bad!)
- Application: `fibonacci()`

Recursion

- **Recursion** is when a function invokes itself
- Classic example: **factorial** (!)
 - $n! = n(n-1)(n-2)(n-3) \dots (3)(2)(1)$
 - $0! = 1$
- Compute **recursively**:
 - **Inductive step**: $n! = n*(n-1)!$
 - **Base case**: $0! = 1$
- Inductive step: **assume** $(n-1)!$ is calculated correctly; then we can find $n!$
- Base case is needed to tell us where to **start**

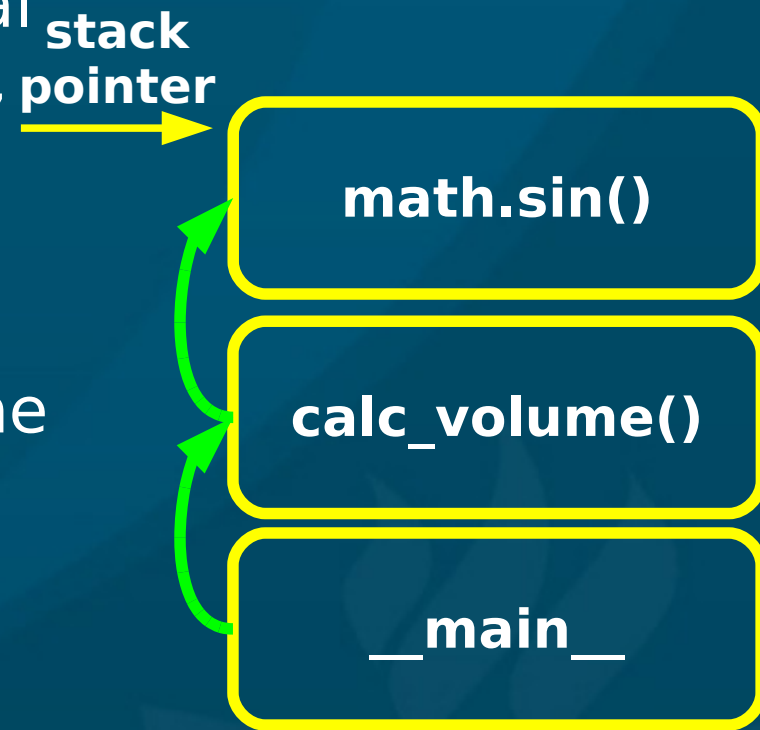
factorial() in Python

```
def factorial(n):  
    """Calculate n!. n should be a positive  
    integer."""  
    if n == 0:                # base case  
        return 1  
    else:                    # inductive step  
        return n * factorial(n-1)
```

- Progress is made each time: `factorial(n-1)`
- Base case prevents **infinite** recursion
- What about `factorial(-1)`? Or `factorial(2.5)`?

The call stack

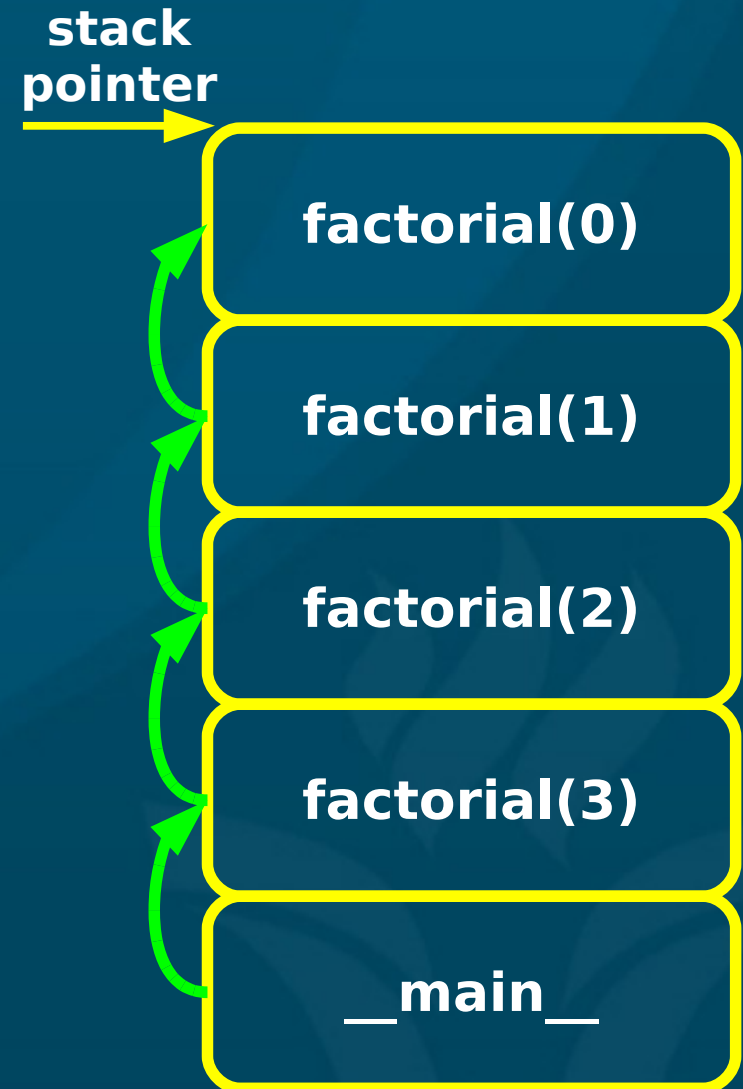
- When a program is running, an area of **memory** is set aside to store local **variables**, the state of the program, **stack pointer** etc.
- When a **procedure** is invoked, the **calling context** is saved, and a new chunk of memory is allocated for the procedure to use: its **stack frame**
- When the procedure finishes, its frame is **released** and control goes back to the calling context
- The **stack pointer** keeps track of what frame is currently running



Call stack for recursion

```
def factorial(n):  
    """Compute the factorial of a  
    positive integer."""  
    if n == 0:  
        return 1  
    else:  
        return n*factorial(n-1)
```

- If there were any **local** variables, each frame would have its own instance of the local variables
- When an error (exception) happens, IDLE shows a **backtrace**: part of the call stack



Tail recursion is bad

- Because the Python runtime must keep all the **stack frames** from previous invocations, recursion is relatively **slow** and uses **memory**.
- **Tail recursion** is when the last computation before returning is a recursive call
 - Better to re-write this using **loops**!

```
def factorial(n):  
    prod = 1  
    for factor in range(1, n):  
        prod *= factor  
    return prod
```

Recursion example: Fibonacci

- **Fibonacci** sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34,...

- Each number is the **sum** of the two previous

def fibonacci(n):

```
    """Compute the n-th Fibonacci number.
```

```
    pre: n should be a positive integer."""
```

```
    if n == 0 or n == 1:                # base case
```

```
        return 1
```

```
    else:                                # inductive step
```

```
        return fibonacci(n-2) + fibonacci(n-1)
```

- Note: very **inefficient** algorithm!

Python type hierarchy (partial)

- **Atomic** types

- Numbers

- ◆ Integers (int, long, bool): **5, 500000L, True**
- ◆ Reals (float) (only double-precision): **5.0**
- ◆ Complex numbers (complex): **5+2j**

- Container (**aggregate**) types

- Immutable sequences

- ◆ Strings (str): **"Hello"**
- ◆ Tuples (tuple): **(2, 5.0, "hi")**

- Mutable sequences

- ◆ Lists (list): **[2, 5.0, "hi"]**

- Mappings

- ◆ Dictionaries (dict): **{"apple": 5, "orange": 8}**

Dictionaries

- Python **dictionaries** are mutable, unsorted containers holding associative key-value pairs
- **Create** a dictionary with curly braces `{}`:
 - ◆ `appleInv = {'Fuji': 10, 'Gala': 5, 'Spartan': 7}`
- **Index** a dictionary using a **key**:
 - ◆ `appleInv['Fuji']` # returns 10
- **Values** can be any object and may mix **types**:
 - ◆ `appleInv['Rome'] = range(3)`
- **Keys** can be any **immutable** type (even tuples!):
 - ◆ `appleInv[('BC', 'Red Delicious')] = 12`