Computational Structures in Data Science

UC Berkeley EECS
Adj. Ass. Prof.
Dr. Gerald Friedland

Lecture #3: Recursion

Go watch Inception!
(Movie about recursion)

February 2nd, 2018 http://inst.eecs.berkeley.edu/~cs88

CS88 news

- Homework will have "Challenge problems"
- Project 1 coming soon!

Site to know: www.stackoverflow.com

- Waitlist: We try to squeeze everybody in!

Computational Concepts today

- Variable Scope (also: see reading)
- Recursion

Remember: Functions

`def` <function name> `(`<argument list>`) :`

`return` expression

```
def concat(str1, str2):
   return str1+str2;

concat("Hello","World")
```

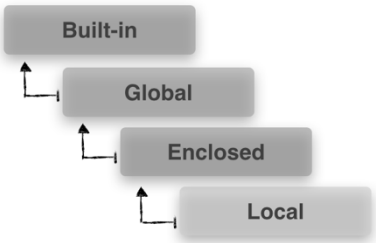
- Generalizes an expression or set of statements to apply to lots of instances of the problem
- A function should *do one thing well*

Variable Scope

When an input is passed to a function, what does the function actually get?
- Internal variables get a *copy* of input values, with the exception of mutable objects

Local variables only exist within the function in which they are defined
- The variables cease to exist when the function ends
- The *scope* of a variable is the part(s) of code where that variable name binding is valid (i.e. where it exists)

Variable Scope (Python)



Variable Scope: Example I

```
i = 1

def foo():
    i = 5
    print(i, 'in foo()')

print(i, '=global')

foo()
```

Output?

1=global
5 in foo()

Variable Scope: Example II

```
a_var = 'global value'

def a_func():
    global a_var
    a_var = 'local value'
    print(a_var, '[ a_var inside
a_func() ]')

print(a_var, '[ a_var outside a_func() ]')
a_func()
print(a_var, '[ a_var outside a_func() ]')
```

Output?

global value [a_var outside a_func()]
local value [a_var inside a_func()]
local value [a_var outside a_func()]

Recursion

re·cur·sion
/riˈkərZHən/

noun MATHEMATICS LINGUISTICS

the repeated application of a recursive procedure or definition.

- a recursive definition.
  plural noun: **recursions**

re·cur·sive
/riˈkərsiv/

adjective

characterized by recurrence or repetition, in particular.

- MATHEMATICS LINGUISTICS
  relating to or involving the repeated application of a rule, definition, or procedure to successive results.
- COMPUTING
  relating to or involving a program or routine of which a part requires the application of the whole, so that its explicit interpretation requires in general many successive executions.

Recursive function calls itself, directly or indirectly

Reminder: Iteration

```
<initialization statements>
for <variables> in <sequence expression>:
  <body statements>

<rest of the program>
```

```
<initialization statements>
while <predicate expression>:
   <body statements>

<rest of the program>
```

```
[ <expr with loop var> for <loop var> in <sequence expr > ]
```


Iteration vs Recursion

For loop:

```
def sum(n):
    s=0
    for i in range(0,n+1):
        s=s+i
    return s
```


Iteration vs Recursion

While loop:

```
def sum(n):
  s=0
  i=0
  while i<n:
      i=i+1
      s=s+i
  return s
```

Iteration vs Recursion

Recursion:

```
def sum(n):
    if n==0:
        return 0
    return n+sum(n-1)
```


Recursion: Pattern

1. Test for simple "base" case
2. Solution in simple "base" case

```
def sum(n):
    if n == 0:
        return 0
    return n + sum(n-1)
```

4. Transform sol'n of simpler problem into full sol'n
3. Assume recursive solution to simpler problem

- **Linear recursion**

Why does it work?

```
sum(3)

# sum(3) => 3 + sum(2)
#              => 3 + sum(2) + sum(1)
#              => 3 + sum(2) + sum(1) + sum(0)
#              => 3 + sum(2) + sum(1) + 0
#              => 3 + sum(2) + 1
#              => 3 + 3
#              => 6
```


How does it work?

- **Each recursive call gets its own local variables**
  - **Just like any other function call**
- **Computes its result (possibly using additional calls)**
  - **Just like any other function call**
- **Returns its result and returns control to its caller**
  - **Just like any other function call**
- **The function that is called happens to be itself**
  - **Called on a simpler problem**
  - **Eventually bottoms out on the simple base case**
- **Reason about correctness "by mathematical induction"**
  - **Solve a base case**
  - **Assuming a solution to a smaller problem, extend it**

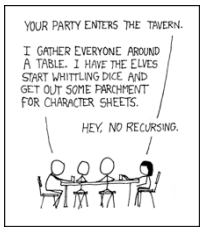
Local variables

```
def sum(n):
    if n==0:
        return 0
    return n+sum(n-1)
```

Each call has its own "frame" of local variables

Sanity Check…

- **Recursion is ■ Iteration (i.e., loops)**
  - **a) more powerful than**
  - **b) just as powerful as**
  - **c) less powerful than**

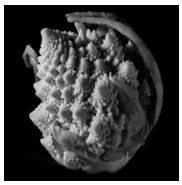


Why Recursion?

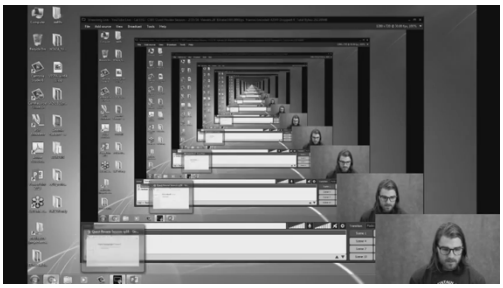
- "After Abstraction, Recursion is probably the 2nd biggest idea in this course"
- "It's tremendously useful when the problem is self-similar"
- "It's no more powerful than iteration, but often leads to more concise & better code"
- "It's more 'mathematical'"
- "It embodies the beauty and joy of computing"
- …

Why Recursion? More Reasons

- Recursive structures exist (sometimes hidden) in nature and therefore in data!
- It's mentally and sometimes computationally more efficient to process recursive structures using recursion.

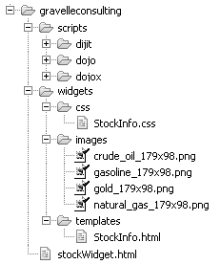


Recursion (unwanted)



Example I

List all items on your hard disk



- Files
- Folders contain
  - Files
  - Folders

Recursion!

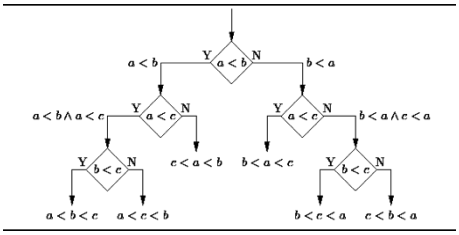
List Files in Python

```
def listfiles(directory):
    content = [os.path.join(directory, x) for x in os.listdir(directory)]

    dirs = sorted([x for x in content if os.path.isdir(x)])
    files = sorted([x for x in content if os.path.isfile(x)])

    for d in dirs:
        print d
        listfiles(d)

    for f in files:
        print f
```

Iterative version about twice as much code and much harder to think about.

Example II

Sort the numbers in a list.



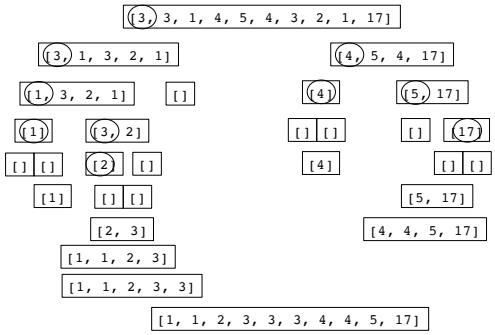
Hidden recursive structure: Decision tree!

Tree Recursion makes Sorting Efficient

Break the problem into multiple smaller sub-problems, and solve them recursively

```
def split(x, s):
    return [i for i in s if i <= x], [i for i in s if i > x]

def qsort(s):
    """Sort a sequence - split it by the first element,
    sort both parts and put them back together."""
    if not s:
        return []
    else:
        pivot = first(s)
        lessor, more = split(pivot, rest(s))
        return qsort(lessor) + [pivot] + qsort(more)

>>> qsort([3,3,1,4,5,4,3,2,1,17])
[1, 1, 2, 3, 3, 3, 4, 4, 5, 17]
```


QuickSort Example



Questions?