Today: Recursion

- Recursive function calls itself, directly or indirectly

Computational Concepts Toolbox

- Iteration:
  - data-driven (list comprehension)
  - control-driven (for statement)
- Higher Order Functions
  - Functions as values
  - Functions with functions as argument
  - Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
  - Function factories – create and return functions

Review: Higher Order Functions

- Functions that operate on functions
- A function
  ```python
def odd(x):
    return x%2
>>> odd(3)
1
```

- A function that takes a function arg
  ```python
def filter(fun, s):
    return [x for x in s if fun(x)]
>>> filter(odd, [0,1,2,3,4,5,6,7])
[1, 3, 5, 7]
```
Review Higher Order Functions (cont)

- A function that returns (makes) a function

```python
def leq_maker(c):
    def leq(val):
        return val <= c
    return leq
```

```python
>>> leq_maker(3)
<function leq_maker.<locals>.leq at 0x1019d8c80>

>>> leq_maker(3)(4)
False

>>> filter(leq_maker(3), [0, 1, 2, 3, 4, 5, 6, 7])
[0, 1, 2, 3]
```
**In words**

- The sum of no numbers is zero
- The sum of $1^2$ through $n^2$ is the sum of $1^2$ through $(n-1)^2$ plus $n^2$.

```python
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```

**Why does it work**

```python
sum_of_squares(3)
# sum_of_squares(3) => sum_of_squares(2) + 3**2
#   => sum_of_squares(1) + 2**2 + 3**2
#   => sum_of_squares(0) + 1**2 + 2**2 + 3**2
#   => 0 + 1**2 + 2**2 + 3**2 = 14
```

**Questions**

- In what order do we sum the squares?
- How does this compare to an iterative approach?

```python
def sum_of_squares(n):
    return sum_of_squares(n-1) + n**2
```

**Tail Recursion**

- All the work happens on the way down the recursion
- On the way back up, just return

```python
def sum_up_squares(i, n, accum):
    if i > n:
        return accum
    else:
        return sum_up_squares(i+1, n, accum + i**2)
```

```python
>>> sum_up_squares(1,3,0)
14
```

**Local variables**

- Each call has its own “frame” of local variables
- What about globals?
- Let’s see the environment diagrams (next lecture)

https://goo.gl/CiFaUJ

```python
def sum_of_squares(n):
    n_squared = n**2
    if n < 1:
        return 0
    else:
        return n_squared + sum_of_squares(n-1)
```
Iteration vs Recursion

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

Iteration vs Recursion

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

Iteration vs Recursion

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

For Homework

```python
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
where fibonacci(1) == fibonacci(0) == 1
```

Another Example

```python
def first(s):
    # Return the first element in a sequence.
    return s[0]

def rest(s):
    # Return all elements in a sequence after the first.
    return s[1:]

def min_r(s):
    # Return minimum value in a sequence.
    if len(s) == 1:
        return first(s)
    else:
        return min(first(s), min_r(rest(s)))
```

Visualization of behavior (print)

```python
In [104]: def min_r(s):
    print('min_r(', s, ') --> ', result)
In [105]: min_r([3, 4, 2, 5, 11])
min_r([3, 4, 2, 5, 11]) --> 2
```

• Recursion over sequence length, rather than number magnitude

• What about sum?
• Don’t confuse print with return value
Trust …

- The recursive “leap of faith” works as long as we hit the base case eventually

What happens if we don’t?

Recursion

- Recursion is...
  
  A) Less powerful than a for loop  
  B) As powerful as a for loop  
  C) As powerful as a while loop  
  D) More powerful than a while loop  
  E) Just different all together

Solution:
C) Any recursion can be formulated as a while loop and any while loop can be formulated as a recursion (with a global variable).

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.

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”  
- …

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.

a < b
a < b < c
b < a

Hidden recursive structure: Decision tree!

Computational Concepts Toolbox

- Data type: values, literals, operations,
- e.g., int, float, string
- Expressions, Call expression
- Variables
- Assignment Statement
- Sequences: tuple, list
- indexing
- Data structures
- Tuple assignment
- Call Expressions
- Function Definition
- Statement
- Conditional Statement

- Iteration:
  - data-driven (list comprehension)
  - control-driven (for statement)
  - while statement
- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
- Assignment of function values
- Recursion

Answers for the Wandering Mind (Holiday Edition)

- How many answers can be maximally responded to by 20 questions (how much data do I need on my game device)?

Assume a number of answer possibilities $b$. This gives $b^{20}$ possible answer paths. In below device: $b=4$ ("unknown", "no", "yes", "sometimes") and $4^{20}=1,099,511,627,776$.

Even if each questions and each answer was only 1 byte long, the device would have to have peta bytes of memory.

Answers for the Wandering Mind (Holiday Edition)

- How can a 20-questions game get away with less?

Different answers lead to the same path (redundancy). For example, making $b$ effectively 2 (instead of 4) results in only $2^{20}=1,048,576$ concepts. The 20-volume Oxford English Dictionary only describes 171,476 ($<4^2$) words. Typically, in our every-day life we deal with about 2000-4000 concepts ($<4^6$).

- How can you make a 20 questions game fail?

Pick a new concept “data science” or chose a random(!) one from the dictionary!

Thoughts for the Wandering Mind

The computer choses a random element $x$ of the list generated by range(0, n). What is the smallest amount of iteration/recursion steps the best ever algorithms needs to guess $x$?

How would the algorithm look like?