Generators, Recursion, and Fractals

1 Generators
- computing a list of Fibonacci numbers
- defining a generator with `yield`
- putting `yield` in the function `fib`

2 Recursive Functions
- computing factorials recursively
- computing factorials iteratively

3 Recursive Images
- some examples
- recursive definition of the Cantor set
- recursive drawing algorithm

MCS 260 Lecture 41
Introduction to Computer Science
Jan Verschelde, 22 April 2016
Generators, Recursion, and Fractals

1 Generators
- computing a list of Fibonacci numbers
- defining a generator with `yield`
- putting `yield` in the function `fib`

2 Recursive Functions
- computing factorials recursively
- computing factorials iteratively

3 Recursive Images
- some examples
- recursive definition of the Cantor set
- recursive drawing algorithm
The Fibonacci numbers are the sequence
\[ 0, 1, 1, 2, 3, 5, 8, \ldots \]
where the next number in the sequence is
the sum of the previous two numbers in the sequence.

Suppose we have a function:

```python
def fib(k):
    """
    Computes the k-th Fibonacci number.
    """
```

and we want to use it to compute the first 10 Fibonacci numbers.
the function `fib`

def fib(k):
    """
    Computes the k-th Fibonacci number.
    """
    if k == 0:
        return 0
    elif k == 1:
        return 1
    else:
        (prevnum, nextnum) = (0, 1)
        for i in range(1, k):
            (prevnum, nextnum) = (nextnum, prevnum + nextnum)
        return nextnum
the main program

def main():
    """
    Prompts the user for a number n and
    prints the first n Fibonacci numbers.
    """
    nbr = int(input('give a natural number n : '))
    fibnums = [fib(i) for i in range(nbr)]
    print(fibnums)

Running at the command prompt $

$ python fibnum.py
give a natural number n : 10
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Problem: with each call to fib, we recompute too much.
Generators, Recursion, and Fractals

1 Generators
- computing a list of Fibonacci numbers
- **defining a generator with** `yield`
- putting `yield` in the function `fib`

2 Recursive Functions
- computing factorials recursively
- computing factorials iteratively

3 Recursive Images
- some examples
- recursive definition of the Cantor set
- recursive drawing algorithm
defining a generator with \texttt{yield}

```python
def counter(value=0):
    ""
    Maintains a counter.
    ""
    count = value
    while True:
        count = count + 1
        yield count
```

If saved in \texttt{show\_yield.py}, then we can do

```python
>>> from show\_yield import counter
>>> mycounter = counter(3)
>>> next(mycounter)
4
>>> next(mycounter)
5
>>>```
using a generator

def main():
    """
    Example of the use of a generator.
    """
    print('initializing counter ...')
    mycounter = counter(3)
    print('incrementing counter ...')
    print(next(mycounter))
    print('incrementing counter ...')
    print(next(mycounter))

if __name__ == "__main__":
    main()
Generators, Recursion, and Fractals

1. Generators
   - computing a list of Fibonacci numbers
   - defining a generator with `yield`
   - putting `yield` in the function `fib`

2. Recursive Functions
   - computing factorials recursively
   - computing factorials iteratively

3. Recursive Images
   - some examples
   - recursive definition of the Cantor set
   - recursive drawing algorithm
def fib(k):
    """
    Computes the k-th Fibonacci number, for k > 1
    """
    (prevnum, nextnum) = (0, 1)
    for i in range(k):
        (prevnum, nextnum) = (nextnum, prevnum + nextnum)
    yield nextnum
the main function

def main():
    """
    Prompts the user for a number n and
    prints the first n Fibonacci numbers.
    """
    nbr = int(input('give a natural number n : '))
    fibgen = fib(nbr)
    fibnums = [next(fibgen) for i in range(nbr)]
    print(fibnums)

Running at the command prompt:

$ python fibyield.py
give a natural number n : 10
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
$

Observe the relabeling.
Generators, Recursion, and Fractals

1 Generators
- computing a list of Fibonacci numbers
- defining a generator with `yield`
- putting `yield` in the function `fib`

2 Recursive Functions
- computing factorials recursively
- computing factorials iteratively

3 Recursive Images
- some examples
- recursive definition of the Cantor set
- recursive drawing algorithm
Computing Factorials Recursively
rule based programming

Let $n$ be a natural number.
By $n!$ we denote the factorial of $n$.

Its recursive definition is given by two rules:

1. for $n \leq 1$: $n! = 1$
2. if we know the value for $(n-1)!$
   then $n! = n \times (n-1)!$

Recursion is similar to mathematical proof by induction:

1. first we verify the trivial or base case
2. assuming the statement holds for all values smaller than $n$ – the induction hypothesis – we extend the proof to $n$
def factorial(n):
    """
    computes the factorial of n recursively
    """
    if n <= 1:
        return 1
    else:
        return n*factorial(n-1)

def main():
    nbr = int(input('give a natural number n : '))
    fac = factorial(nbr)
    print('n! = ', fac)
    print('len(n!) = ', len(str(fac)))

Intro to Computer Science (MCS 260)
generators and recursion
L-41  22 April 2016  14 / 36
tracing recursive functions

Calling \texttt{factorial} for \( n = 5 \):

\begin{align*}
\text{factorial}(5) \text{ call #0: call for } n-1 &= 4 \\
\text{factorial}(4) \text{ call #1: call for } n-1 &= 3 \\
\text{factorial}(3) \text{ call #2: call for } n-1 &= 2 \\
\text{factorial}(2) \text{ call #3: call for } n-1 &= 1 \\
\text{factorial}(1) \text{ call #4: base case, return } 1 \\
\text{factorial}(2) \text{ call #3: returning } 2 \\
\text{factorial}(3) \text{ call #2: returning } 6 \\
\text{factorial}(4) \text{ call #1: returning } 24 \\
\text{factorial}(5) \text{ call #0: returning } 120
\end{align*}

Computes in returns:

\texttt{return 1, 1*2, 1*2*3, 1*2*3*4, 1*2*3*4*5}
Generators, Recursion, and Fractals

1 Generators
- computing a list of Fibonacci numbers
- defining a generator with `yield`
- putting `yield` in the function `fib`

2 Recursive Functions
- computing factorials recursively
- computing factorials iteratively

3 Recursive Images
- some examples
- recursive definition of the Cantor set
- recursive drawing algorithm
running the recursive factorial

Long integers are no problem, but ...

$ python factorial.py
give a natural number n : 79
n! = 894618213078297528685144171539831652
069808216779571907213868063227837990693501
86053336181084101017600000000000000000
len(n!) = 117

Exploiting Python long integers:

$ python factorial.py
give a number : 1234
...
RuntimeError: maximum recursion depth exceeded

An exception handler will compute $n!$ iteratively.
stack of function calls

The execution of recursive functions requires a stack of function calls.

For example, for $n = 5$, the stack grows like

```
factorial(1) call #4: base case, return 1
factorial(2) call #3: returning 2
factorial(3) call #2: returning 6
factorial(4) call #1: returning 24
factorial(5) call #0: returning 120
```

New function calls are pushed on the stack. Upon return, a function call is popped off the stack.
computing factorials iteratively in an exception handler

def factexcept(n):
    """
    when the recursion depth is exceeded
    the factorial of n is computed iteratively
    """
    if n <= 1:
        return 1
    else:
        try:
            return n*factexcept(n-1)
        except RuntimeError:
            f = 1
            for i in range(2, n+1):
                f = f*i
            return f
Generators, Recursion, and Fractals

1. Generators
   - computing a list of Fibonacci numbers
   - defining a generator with `yield`
   - putting `yield` in the function `fib`

2. Recursive Functions
   - computing factorials recursively
   - computing factorials iteratively

3. Recursive Images
   - some examples
   - recursive definition of the Cantor set
   - recursive drawing algorithm
A Cantor Set
A Koch Curve
A Koch Flake
A Sierpinski Gasket
Generators, Recursion, and Fractals

1. Generators
   - computing a list of Fibonacci numbers
   - defining a generator with `yield`
   - putting `yield` in the function `fib`

2. Recursive Functions
   - computing factorials recursively
   - computing factorials iteratively

3. Recursive Images
   - some examples
   - recursive definition of the Cantor set
   - recursive drawing algorithm
The Cantor Set

The Cantor set is defined by three rules:

1. take the interval $[0, 1]$;
2. remove the middle part third of the interval;
3. repeat rule 2 on the first and third part.

The Cantor set is infinite, to visualize at level $n$:

- $n = 0$: start at $[0, 1]$;
- $n > 0$: apply rule 2 $n$ times.
GUI for a Cantor Set

---

### Cantor Set Generator

This GUI allows you to visualize the Cantor set and its iterations. The Cantor set is a mathematical set of points on a line segment that is constructed by iteratively removing the middle third of each line segment. Each iteration is represented by a level, and the length of each segment is reduced by a factor of 3.

In the GUI, you can:
- **Select the level (0-6)** to display the Cantor set at that stage.
- **Click on 'clear canvas'** to reset the display to the initial state.

---

**Level 0**
- Initial segment

**Level 1**
- Remains intact

**Level 2**
- Removes the middle third from the remaining segments

**Level 3**
- Further refinement of the middle third

**Level 4**
- Continuation of the process

**Level 5**
- Additional iterations

**Level 6**
- Final iteration of the Cantor set generation

---

This visualization is an example of how recursion can be used to create complex structures from simple rules.
def __init__(self, wdw, N):
    """
    A Cantor set with N levels.
    """

def draw_set(self, val):
    """
    Draws a Cantor set.
    """

def clear_canvas(self):
    """
    Clears the entire canvas.
    """
def __init__(self, wdw, N):
    
    A Cantor set with N levels.
    
    wdw.title('a cantor set')
    self.dim = 3**N+20
    self.n = IntVar()
    self.scl = Scale(wdw, orient='horizontal',
                    from_=0, to=N, tickinterval=1,
                    length=self.dim, variable=self.n,
                    command=self.draw_set)
    self.scl.grid(row=0, column=0)
    self.scl.set(0)
    self.cnv = Canvas(wdw, width=self.dim,
                      height=self.dim/3, bg='white')
    self.cnv.grid(row=1, column=0)
    self.btt = Button(wdw, text="clear canvas", 
                      command=self.clear_canvas)
    self.btt.grid(row=2, column=0)
The method `clear_canvas()` is triggered by a Button.

```python
def clear_canvas(self):
    """
    Clears the entire canvas.
    """
    self.cnv.delete(ALL)
```

The method `DrawSet()` is triggered by a Scale.

```python
def draw_set(self, val):
    """
    Draws a Cantor set.
    """
    nbr = int(val)
    self.cantor(10, self.dim-10, 30, val, nbr)
```

The method `cantor` is recursive.
Generators, Recursion, and Fractals

1. Generators
   - computing a list of Fibonacci numbers
   - defining a generator with `yield`
   - putting `yield` in the function `fib`

2. Recursive Functions
   - computing factorials recursively
   - computing factorials iteratively

3. Recursive Images
   - some examples
   - recursive definition of the Cantor set
   - recursive drawing algorithm
def cantor(self, lft, rgt, hgt, txt, lvl):
    """
    Draws a line from lft to rgt, at height hgt
txt is a string, int(txt) equals the number
of times the middle third must be removed
lvl is level of recursion, start at lvl = int(txt)
    """

The parameters lft, rgt, and hgt define
the line segment from (lft,hgt) to (rgt,hgt).

The parameter txt is the value passed via the Scale,
as text string, txt is also put on Canvas.

Initially: lvl = int(txt).
With every recursive call, lvl is decremented by 1.
a recursive drawing algorithm

The `lvl` in `cantor(self, lft, rgh, hgt, txt, lvl)` controls the recursion.

At `lvl = 0`, the line segment from `(lft, hgt)` to `(rgh, hgt)` is drawn.

For `lvl > 0`, we compute left and right limit of the middle third of `[lft, rgh]`, respectively denoted by `nlf` and `nrg` as

\[
\begin{align*}
\text{nlf} &= lft + (rgh - lft) / 3 = (2lft + rgh) / 3 \\
\text{nrg} &= rgh - (rgh - lft) / 3 = (lft + 2rgh) / 3
\end{align*}
\]

Then we make two recursive calls:

\[
\begin{align*}
\text{self.cantor(lft, nlf, hgt+30, txt, lvl-1)} \\
\text{self.cantor(nrg, rgh, hgt+30, txt, lvl-1)}
\end{align*}
\]
def cantor(self, lft, rgt, hgt, txt, lvl):
    "..."
    if lvl == 0:  # draw line segment
        self.cnv.create_line(lft, hgt, rgt, hgt, width=2)
    else:
        nlf = (2*lft+rgt)//3
        nrg = (lft+2*rgt)//3
        self.cantor(lft, nlf, hgt+30, txt, lvl-1)
        self.cantor(nrg, rgt, hgt+30, txt, lvl-1)
    if lvl == int(txt):  # put text string
        xctr = self.dim//2
        if txt == '0':
            self.cnv.create_text(xctr, hgt-10, text=txt)
        else:
            self.cnv.create_text(xctr, hgt+lvl*30, 
                text=txt)
some closing observations

What comes next?

- mcs 275: programming tools and file management
- mcs 320: symbolic computation
- mcs 360: data structures