Computer Science 1001.py

Lecture 10[†]: More Recursion
Lambda Expressions and Higher Order Functions

Instructor: Benny Chor

Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University

Fall Semester, 2011/12
http://tau-cs1001-py.wikidot.com

[†] © Benny Chor

Lecture 9 Highlights: Recursion

- Recursion is a powerful control mechanism.
- Often follows recursive definitions.
- In many cases, it is easy and natural to code recursively.
- Recursive definitions with simple, highly concise code can hide terrible (exponential time) performance.
- In some cases, techniques like memorizing/dynamic programming lead to more efficient code.
- In some cases, recursion can be fully replaced by an iterative approach, and furthermore, substantial saving in memory may be possible.
- In other cases the recursive implementation is both natural and efficient or even optimal, *e.g.* `quicksort` (efficient) or `HanoiTowers` (optimal).

Lecture 9, Python Highlights

- Recursive functions.
- Recursion depth limit.

- `enumerate`
- List comprehension.

Reflections: Memoization, Iteration, Memory Reuse

In the Fibonacci numbers example, all these techniques proved relevant and worthwhile performance wise. These techniques won't always be applicable for every recursive implementation of a function.

Consider `quicksort` as a specific example. In any specific execution, we never call quicksort on the same set of elements more than once (think why this is true).

So memoization is not applicable. And replacing recursion by iteration, even if applicable, may not be worth the trouble and surely will result in less elegant and possibly more error prone code.

Even if these techniques are applicable, the transformation is often not automatic, and if we deal with small instances where performance is not an issue, such optimization may be a waste of effort.

Not for the Soft At Heart: the Ackermann Function

This recursive function, invented by the German mathematician Wilhelm Friedrich Ackermann (1896–1962), is defined as following:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0\ . \end{cases}$$

This is a total recursive function, namely it is defined for all arguments (pairs of non negative integers), and is computable (it is easy to write Python code for it). However, it is what is known as a non primitive recursive function, and one manifestation of this is its huge rate of growth.

You will meet the inverse of the Ackermann function in the data structures course as an example of a function that grows to infinity very very slowly.

Ackermann function: Python Code

Writing down Python code for the Ackermann function is easy – just follow the definition.

```
def ackermann(m,n):
    # Ackermann function
    if m==0:
        return n+1
    elif m>0 and n==0:
        return ackermann(m-1,1)
    else:
        return ackermann(m-1,ackermann(m,n-1))
```

However, running it with $m \geq 4$ and any positive n causes run time errors, due to exceeding Python's maximum recursion depth. Even `ackermann(4,1)` causes such a outcome.

Ackermann function and Python Recursion Depth

However, running it with $m \geq 4$ and any positive n causes run time errors, due to exceeding Python's maximum recursion depth. Even `ackermann(4,1)` causes such outcome.

As we saw earlier, Python has a default limit of 1,000 on recursion depth. You can import the Python `sys` library, find out what the limit is, and also change it.

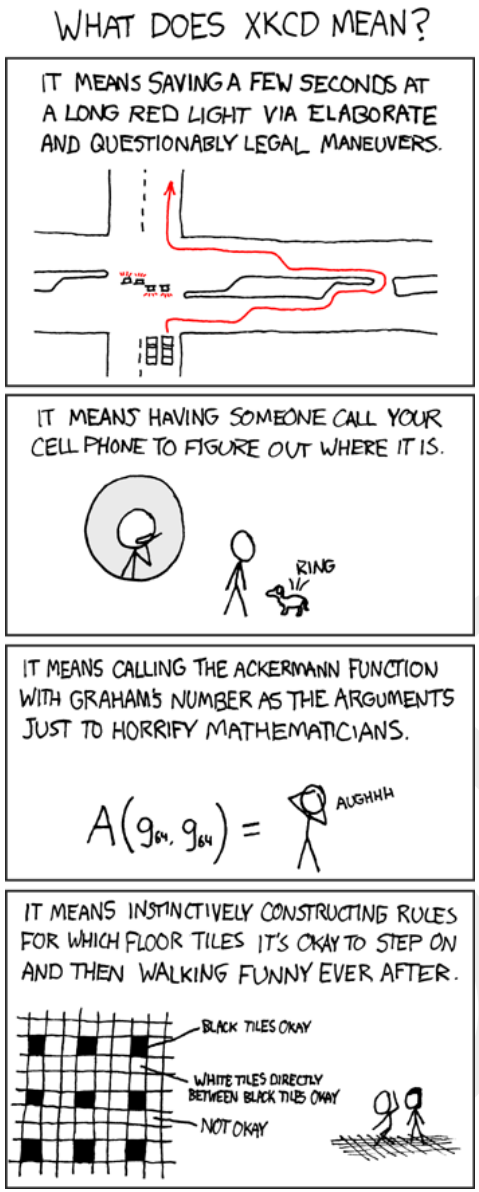
```
>>> import sys
>>> sys.getrecursionlimit()     # find recursion depth limit
>>> sys.setrecursionlimit(100000)     # change limit
```

However, even with this much larger limit, 100,000, only on Linux `ackermann(4,1)` ran to completion (returning just `65,533`, by the way). On an 8GB RAM machine running either MAC OSX 10.6.8 or Windows 7, it simply crashed (reporting "segmentation fault").

Adventurous? `ackermann(4,2)` will exceed any recursion depth that Python will accept, and cause your execution to crash.

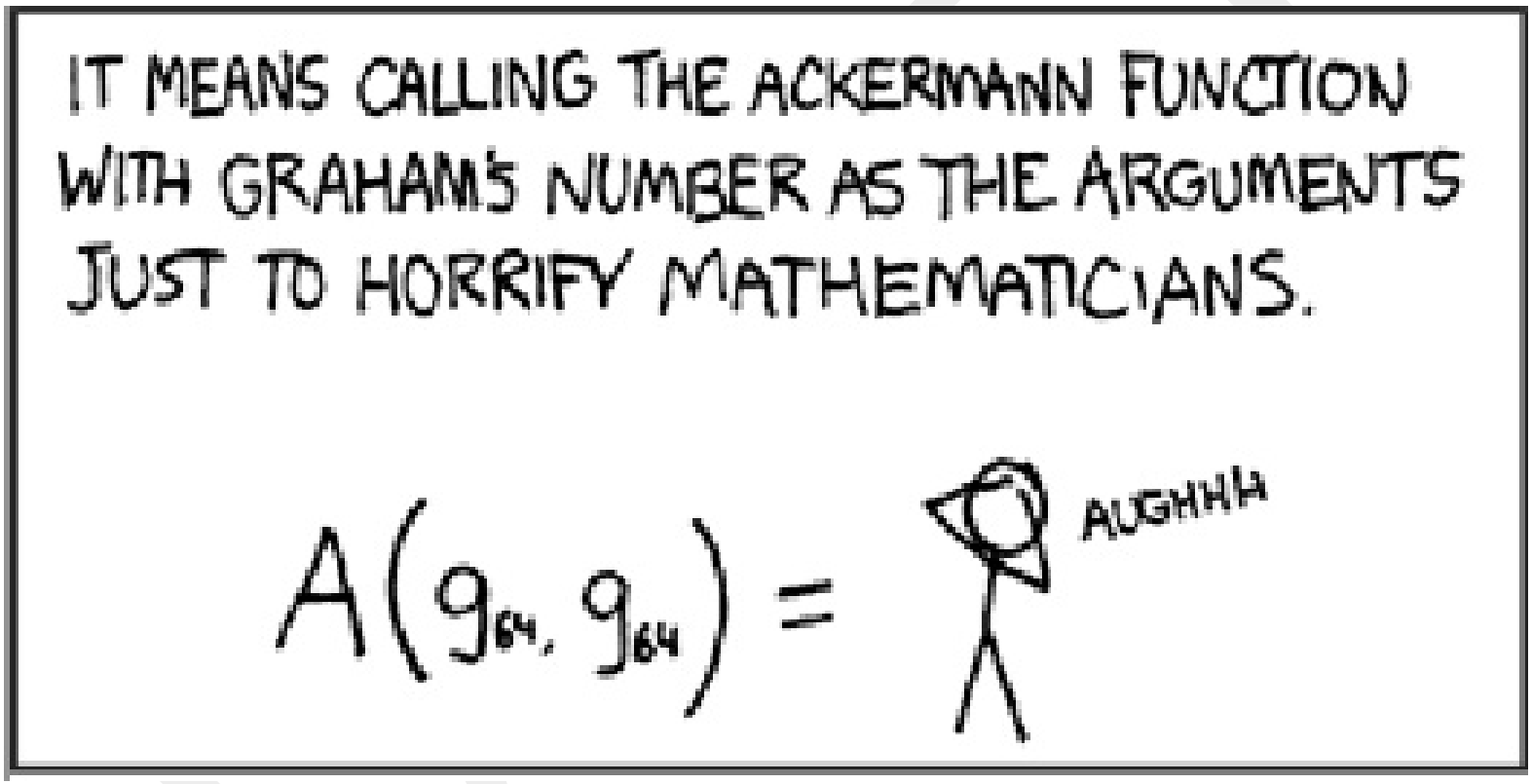
This is what happens with such rapidly growing functions.

XKCD View of the Ackermann Function



(the third entry relates to Ackermann's function)

A Close Up View



(also taken from http://xkcd.com/207/)

Reflections: Towers of Hanoi

Consider, again, the code for solving the Towers of Hanoi puzzle.

```
def HanoiTowers(start,via,target,n):
    """ returns a list of disks moves as to move a stack
    of n disks from rod "start" to rod "target" employing
    intermidiate rod "via" """
    if n==0:
        return []
    else:
        return HanoiTowers(start,target,via,n-1) \
          + [str.format("disk {} from {} to {}", n, start, target)]
          + HanoiTowers(via,start,target,n-1)
```

We claimed that the number of moves required to solve an instance with n disks is $H(n) = 2^n - 1$. Our program generates such a list of disk moves. It runs in $O(H(n))$ time. The recursion depth here is just $O(n)$ (but the width of the recursion is exponential in n).

Towers of Hanoi Nightmare

Suppose your partner[‡] wakes you in the middle of the night after having a terrible dream. A monster demanded to know what the $3^{97} + 19$ move in an $n = 200$ disk Towers of Hanoi puzzle is, or else ....

Having seen and even understood the material in this class, you quickly explain to your frightened partner that either expanding all $H(200) = 2^{200} - 1$ moves, or even just the first $3^{97} + 19$, is out of computational reach in any conceivable future, and the monster should try its luck elsewhere.

Your partner is not convinced.

You eventually wake up and realize that the fastest way for sleeping the rest of the night is to solve this new problem. The first step towards taming the monster is to give the new problem a name: `HanoiMove(start,via,target,n,k)`.

[‡] we are very careful to be gender and sexual inclination neutral here

Feeding the Towers of Hanoi Monster

The solution to `HanoiTowers(start,via,target,n)` consisted of three (unequal) parts, requiring $2^n - 1$ steps altogether:

In the first part, which takes $2^{n-1} - 1$ steps, we move $n - 1$ disks.
In the second part, which takes exactly one step, we move disk number n.
In the last part, which again takes $2^{n-1} - 1$ steps, we move $n - 1$ disks.

The monster wants to know what is move number k out of $2^n - 1$ steps.
If $k = 2^{n-1}$ then this is the "middle step", and we know exactly what this is.

But what do we do if $k \neq 2^{n-1}$?

Well, we think recursively:

Feeding the Towers of Hanoi Monster, cont.

The solution to `HanoiTowers(start,via,target,n)` consisted of three (unequal) parts, requiring $2^n - 1$ steps altogether:
But what do we do if $k \neq 2^{n-1}$?

Well, we think recursively:

If $k < 2^{n-1}$ then this is move number k in an $n-1$ disks problem (part one of the n disks problem).

If $k > 2^{n-1}$ then this is move number $k - 2^{n-1}$ in an $n-1$ disks problem (part three of the n disks problem).

We should keep in mind that the roles of the rods in these smaller subproblems will be permuted.

Recursive Monster Code

```
def hanoi_move(start,via,target,n,k):
    """ finds the k-th move in Hanoi Towers instance
    with n disks """
    if n<=0:
        return "zero or fewer disks"
    elif k<=0 or k>=2**n or type(k)!=int:
        return "number of moves is illegal"
    elif k==2**(n-1):
        return str.format("disk {} from {} to {}", n, start, target
    elif k<2**(n-1):
        return hanoi_move(start,target,via,n-1,k)
    else:
        return hanoi_move(via,start,target,n-1,k-2**(n-1))
```

We first test it on some small cases, which can be verified by running the HanoiTowers program. Once we are satisfied with this, we solve the monster's question (and get back to sleep).

```
>>> hanoi_move("A","B","C",4,8)
'disk 4 from A to C'
>>> hanoi_move("A","B","C",4,4)
'disk 3 from A to B'
>>> hanoi_move("A","B","C",200,3**97+19)
'disk 2 from B to A'      # saved!
```

Recursive Monster Solution: Binary Search

The recursive `HanoiMove(start,via,target,n,k)` makes at most one recursive call.

The way it "homes" on the right move employs the already familiar technique of binary search: It first determines if move number k is exactly the middle move in the n disk problem. If it is, then by the nature of the problem it is easy to exactly determine the move.

If not, it determines if the move is in the first half of the moves' sequence ($k < 2^{n-1}$) or in the second half ($k > 2^{n-1}$), and makes a recursive call with the correct permutation of rods.

The execution length is linear in n (and not in 2^n, the length of the sequence of moves).

Binary Search

We have already seen binary search and realized it is widely applicable (not only when monsters confront you at night). We can use binary search when we look for an item in a huge space, in cases where that space is structured so we could tell if the item is

1. right at the middle,
2. in the top half of the space,
3. or in the lower half of the space.

In case (1), we solve the search problem in the current step. In cases (2) and (3), we deal with a search problem in a space of half the size.

In general, this process will thus converge in a number of steps which is $\log_2$ of the size of the initial search space. This makes a huge difference. Compare the performance to going linearly over the original space of $2^n - 1$ moves, item by item.

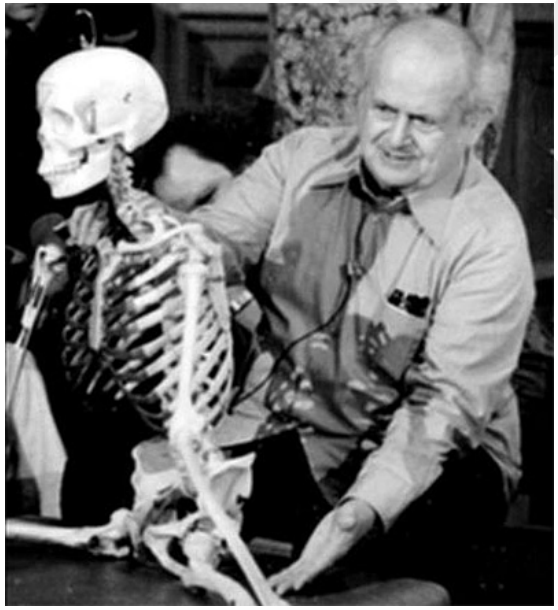
Binary Search (and the Feldenkrais Method)

Quoting (again) Dr. Moshé Feldenkrais (1904-1984), his method "...makes the impossible possible, the possible comfortable, and the comfortable pleasant."

Let us borrow Dr. Feldenkrais' description to binary search.

In the case of searching an ordered list, we could classify binary search as making the possible comfortable.

Here, the search space is so huge that binary search is definitely making the impossible possible.



Replacing Iteration with Recursion

Replacing Iteration withs Recursion

Suppose you wanted to compute $\sum_{k=1}^{n} k$, which in Python is simply `sum(range(1,n+1))`. However, for some reason, the lunatic course staff insists that you do not use `for`, `while`, or `sum`.

What would you do?

Well, luckily you have just learned about recursion.

```
def recursive_sum(n):
    """ computes 1+2+...+n recursively """
    if n==1:
        return 1
    else:
        return n+recursive_sum(n-1)
```

This solution is fine, but much less efficient than the iterative solution. Furthermore, as soon as n approaches, say, 1000, you will hit Python's recursion limit and crash. Of course you can set a higher recursion limit, but even $n = 1000000$ won't be feasible, while it is a piece of cake for `sum`.

Tail Recursion

We modify the code such that the recursive call occurs exactly at the end of the function body, and furthermore it is a pure recursive call (the result of the recursive call is returned AS IS as the result of the current call, and is not mixed with other operations such as the + above).

To do this, we introduce an auxiliary argument, `partial`.

```
def tail_sum(n,partial):
    """ computes partial+(1+2+...+n) using tail recursion """
    if n==1:
        return partial+1
    else:
        return tail_sum(n-1,partial+n)
```

Note that setting a default value, say `partial=0`, will lead to an incorrect result (try to figure out why!).

Tail Recursion, Execusion

```
>>> tail_sum(4)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    tail_sum(4)
TypeError: tail_sum() takes exactly 2 positional arguments (1 given

>>> tail_sum(4,0)
10
>>> tail_sum(4,33)
43
```

Tail Recursion: Automatic Removal of Recursion

```
def tail_sum(n,partial):
    """ computes partial+(1+2+...+n) using tail recursion """
    if n==1:
        return partial+1
    else:
        return tail_sum(n-1,partial+n)
```

A smart compiler can automatically convert this code into a more efficient one, employing no recursive calls at all: When reaching the tail, no new function frame is built, Instead, the function is "freshly invoked" (no history kept) with the parameters `n-1,partial+n`.

Recursion of that form is called tail recursion. This optimization is frequently used in functional languages (including Scheme) where recursion is the main control mechanism.

Recursion in Other Programming Languages

Python, C, Java, and most other programming languages employ recursion as well as a variety of other flow control mechanisms.

By way of contrast, all LISP dialects (including Scheme) use recursion as their major control mechanism. We saw that recursion is often not the most efficient implementation mechanism.

Taken together with the central role of eval in LISP, this may have prompted the following statement, attributed to Alan Perlis of Yale University (1922-1990): "LISP programmers know the value of everything, and the cost of nothing".

λ Expressions (and λ Calculus)

The λ (lambda) calculus was invented by Alonzo Church (1903-1995). Church was one of the great mathematicians and logicians of the twentieth century. Lambda calculus was one of several attempts to capture a mathematical notion of computing, long before actual computers existed.

(photo from Wikipedia)

- You will meet him (Church) at least once more in your studies, when you get to the famous Church thesis in the Computational Models course.
- In the current context, we will concentrate on λ expressions: These have the form $\lambda\ x_1 \ldots x_k$: expression.

λ Expressions, cont.

- In the current context, we will concentrate on λ expressions: These have the form $\lambda \quad x_1 \ldots x_k$: expression .
- Such a λ expression represents an "anonymous function".
- The arguments are the $x_1 \ldots x_k$ $(k \geq 0)$.
- The expression on the right is the body of the function, which is to be executed with the actual values supplied upon calling the function.
- This construct does not have any explicit name. Thus it is an "anonymous" function.
- The function can be applied and executed like any other function.

λ Expressions: A few Examples

Given the three coordinates x, y, z in Euclidean 3D space, the length of the vector going from $(0, 0, 0)$ to (x, y, z) is $\sqrt{x^2 + y^2 + z^2}$.
In Python, this can be defined as following:

```
euclid = lambda x,y,z: (x**2+y**2+z**2)**0.5
```

If you evaluate it, the Python Shell will inform you that this is a function

```
>>> euclid
<function <lambda> at 0x170b6a8>
```

We can also define an anonymous, nameless function

```
>>> lambda x,y,z: (x**2+y**2+z**2)**0.5
<function <lambda> at 0x170b150>
```

This function can subsequently be applied (notice the extra parenthesis) to yield, say, $\sqrt{2^2 + 3^2 + 4^2} = \sqrt{29} \approx 5.38$

```
>>> (lambda x,y,z: (x**2+y**2+z**2)**0.5)(2,3,4)
5.385164807134504
```

Why Should We Care?

We saw we could define

```
euclid = lambda x,y,z: (x**2+y**2+z**2)**0.5
```

and invoke it by calling, *e.g.*

```
euclid(2,3,4)
```

But then

```
def euclid(x,y,z):
    return (x**2+y**2+z**2)**0.5
```

does exactly the same.

So what are these weird, cumbersome λ expression good for in our context?

Examples: "Numerical" Differentiation and Integration

- Differentiation and integration (developed independently by math giants Isaac Newton and Gottfried Leibniz in the late 17th century) are the two fundamental operators in calculus.
- Given a (smooth) real valued function $f : \mathbb{R} \mapsto \mathbb{R}$, its derivative $f' : \mathbb{R} \mapsto \mathbb{R}$ at point x is defined (intuitively) as the slope of f at the point y.
- Given a (not too wild) real valued function $f : \mathbb{R} \mapsto \mathbb{R}$, its definite integral $\int_a^b f(x)dx$ between points a, b is defined (intuitively) as the signed area between the curve of $f(x)$ and the x-axis.

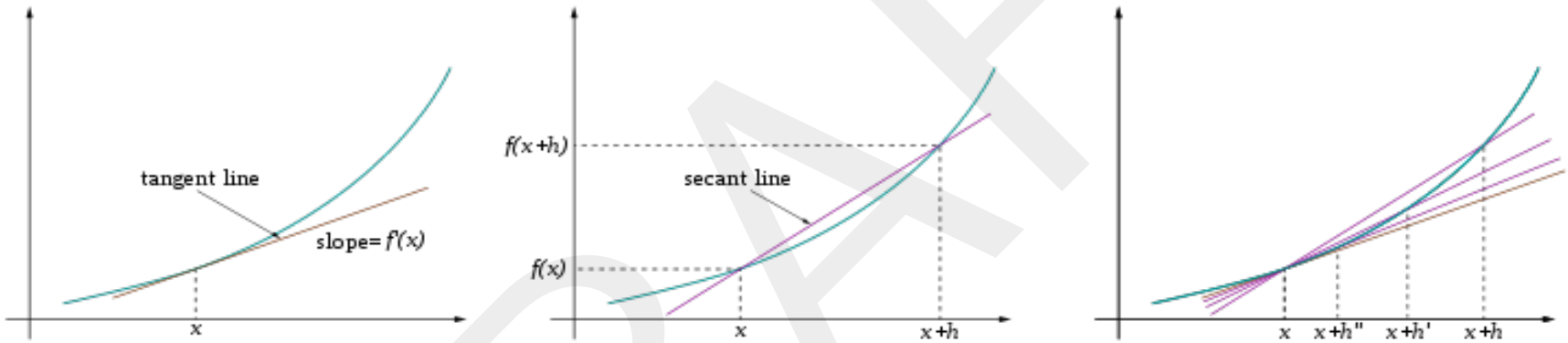
Function Operators

- Both differentiation and integration are function operators. Given the function $f : \mathbb{R} \mapsto \mathbb{R}$, these operators define two other functions (the derivative, and the integral).
- By the fundamental theorem of calculus, integration and differentiation are inverse operators.
- We want to define, say, a Python function `diff` that receives a function, f, as its input argument and produces the function f' (the derivative of f) as its returned value.

The Derivative as a "High Order Operator"

- As you surely recall, the derivative of f at point x is defined by

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} .$$



(drawings from Wikipedia.)

- We want to define a Python function `diff` that receives a function, f, as its input argument and produces the function f' (the derivative of f) as its returned value.
- So unlike "regular functions", this `diff` is a "high order function".
- However, for Python this is nothing unusual.

Implementing Derivatives in Python

- We will make the simplifying assumption that in order to determine the value of the derivative at a point, it suffices to take a small enough h.
- Furthermore, we'll assume that $h = 0.001$ is small enough (this suffices for most functions we'll run into, and you can take even smaller values if you prefer).
- The definition of `diff` employs an anonymous function, defined in terms of a λ expression.

Implementing Derivatives in Python

The definition of diff employs an anonymous function, defined in terms of a λ expression.

```
def diff(f):
    h=0.001
    return (lambda x: (f(x+h)-f(x))/h)
```

```
>>> def f(x):
return x**3

>>> diff(f)(2)  # the derivative of f at y is 3*(y**2)
12.006000999997823   # 3*2**2=12 expected
```

We now compute the derivative of the sine function at the point $\pi/2$.

```
>>> import math
>>> cos=diff(math.sin)
>>> cos(math.pi/2) # cos(pi/2) should be approx. 0
-0.0004999999583255033
>>> math.cos(math.pi/2) # compare to "native Python value"
6.123233995736766e-17
```

Alternative Implementation of Derivatives in Python

- It is possible to give an alternative definition for `diff`, one which avoids the use of a λ expression.

```
def diff2(f):
    h=0.001
    def df(x):
        return(f(x+h)-f(x))/h
    return df
```

- This works fine, but it defines a new function, `df`, whose name is never used elsewhere.
- We, and many others, consider the definition via λ expression more elegant.
- However, this is a matter of taste. Operationally, both are equivalent.

Highly Variable Functions

- So far, we assumed that $h = 0.001$ is small enough for our needs.
- This may be OK in most cases, but for highly variable functions, the outcome may be very inaccurate.
- As a specific (and somewhat artificial) example, consider the function sin_by_million$(x) = \sin(10^6 \cdot x)$. At the point $x = 0$, its derivative is $10^6 \cdot \cos(0) = 10^6$, so

  `diff(sin_by_million)(0)` should be approximately 10^6

```
def sin_by_million(x):
    return math.sin(10**6*x)

>>> diff(sin_by_million)(0)
826.8795405320026
```

826.8795405320026 is, ahhm, not even close to $10^6 = 1,000,000$. The reason for this discrepancy is that $h = 0.001$ is usually small enough, but it is way too big for sin_by_million.

Implementation and Default Parameters' Values

826.8795405320026 is, ahhm, not even close to $10^6 = 1,000,000$. The reason for this discrepancy is that $h = 0.001$ is usually small enough, but it is way too big for sin_by_million.

We already saw that Python provides a mechanism of default values for parameters.

This lets us use a predetermined value as a default parameter, yet use different values when we deem it necessary. Note that the original parameter name has to be explicitly specified to use such different value value.

```
def diff_param(f,h=0.001):
    # when h not specified, default value h=0.001 is used
    return(lambda x: (f(x+h)-f(x))/h)
```

Generalized Implementation and Default Parameters

```
def diff_param(f,h=0.001):
    # when h not specified, default value h=0.001 is used
    return(lambda x: (f(x+h)-f(x))/h)
```

We can now apply this mechanism with different degrees of resolution.

```
>>> diff_param(sin_by_million)(0)
826.8795405320026       # no h specified - default h used
>>> diff_param(sin_by_million,h=0.001)(0)
826.8795405320026       # parameter equals the default h
>>> diff_param(sin_by_million,h=0.00001)(0)
-54402.11108893698      # smaller and smaller  h
>>> diff_param(sin_by_million,h=0.0000001)(0)
998334.1664682814       # better and better accuracy for derivative
>>> diff_param(sin_by_million,h=0.000000001)(0)
999999.8333333416
>>> diff_param(sin_by_million,h=0.000000000001)(0)
999999.9999998333  # indeed almost 1000000, as expected
```

Recall that real numbers (type `float` in Python) have a limit on accuracy. A value that is too small will be interpreted as zero.

Additional Examples

```
def square(x):
    return(x**2)
```

`square` is surely differentiable, and its derivative at the point x is $2x$. We can apply our derivative operator to it.

```
>>> diff(square)(2)
4.00099999999699
>>> diff(square)(3)
6.00099999999479
>>> diff(square)(5)
10.00100000002591
```

Likewise, we define `penta`, whose derivative at the point x is $5x^4$.

```
def penta(x):
    return(x**5)

>>> diff(penta)(2)
80.0800400099888     # 5*2**4=80
>>> diff(penta)(5)
3126.25025002626     # 5*5**4=3125
>>> diff_param(penta,h=0.0000001)(5)
3125.000134787115
```