Computer Science 1001.py, Lecture 7
Higher Order Functions and Lambda Expressions
Numeric Derivative and Integral
Floating Point Arithmetic
Finding Zeroes of Real Functions

Instructors: Benny Chor, Amir Rubinstein
Teaching Assistants: Yael Baran, Michal Kleinbort

Founding Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University
Spring Semester, 2015
http://tau-cs1001-py.wikidot.com

Lecture 6 Highlights

- Basic algorithms
  - Sorting.
  - Merging sorted lists.

- Measuring the Complexity of Algorithms
  - Big O notation.
  - Worst case and best case.
  - Tractable and intractable computational problems.

A Reminder (`Mon = Wed`)

Tel Aviv University authorities have decreed* that Monday, March 30, 2015 is Wednesday, April 1, 2015.

This is not an April Fools' Day hoax!

It means the Lecture 8 of our course will be delivered on Monday, March 30 (13–15 and 15–17).
It also means Monday's recitation will not take place.

By the same token, `Tues = Thurs` this week.

*decree (noun): an official order issued by a legal authority.

Time Complexity - What is tractable in Practice?

- A polynomial-time algorithm is good.
- An exponential-time algorithm is bad.
- n^{100} is polynomial, hence good.
- $2^{n/100}$ is exponential, hence bad.

Yet for input of size $n = 4000$, the n^{100} time algorithm takes more than 10^{35} centuries on the above mentioned machine, while the $2^{n/100}$ runs in just under two minutes.

Time Complexity – Advice

- Trust, but check!

  Don't just mumble "polynomial-time algorithms are good", "exponential-time algorithms are bad" because the lecturer told you so.
- Asymptotic run time and the O notation are important, and in most cases help clarify and simplify the analysis.
- But when faced with a concrete task on a specific problem size, you may be far away from "the asymptotic".
- In addition, constants hidden in the O notation may have unexpected impact on actual running time.

Time Complexity – Advice (cont.)

- When faced with a concrete task on a specific problem size, you may be far away from "the asymptotic".
- In addition, constants hidden in the O notation may have unexpected impact on actual running time.

- We will employ both asymptotic analysis and direct measurements of the actual running time.
- For direct measurements, we will use either the `time` package and the `time.clock()` function.
- Or the `timeit` package and the `timeit.timeit()` function.
- Both have some deficiencies, yet are highly useful for our needs.

And Now to Something Completely Different: Lambda Expressions

The sun was shining on the sea,
Shining with all his might:
He did his very best to make
The billows smooth and bright –
And this was odd, because it was
The middle of the night.

The moon was shining sulkily,
Because she thought the sun
Had got no business to be there
After the day was done –
"It's very rude of him," she said,
"To come and spoil the fun!"

Through the Looking-Glass and What Alice Found There: Lewis Carroll, 1871.

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

- Many of you have seen (or will see) the λ notation in the discrete mathematics course.
- 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, cont.

- In the current context, we will concentrate on λ expressions: These have the form $\lambda \quad x_1 \ldots x_k$: expression .
- The expression must be "simple". It cannot include, for example, definition of a function (`def`) or iterations (`for` or `while`).
- It may include `if` or `print`.
- In general, such expressions are OK iff they have a value, and can be the right hand side of an assignment.

λ Expressions: A Few Examples

```
>>> lambda n: sum(range(n))  # kosher
<function <lambda> at 0x102f8e628>

>>> lambda n: sum=0; for i in range(n): sum=sum+i  # not kosher
SyntaxError: invalid syntax

>>> lambda x: x if x>0 else -x   # kosher
<function <lambda> at 0x102f8ef30>

>>> lambda : print("gidday")    # kosher, but not very useful
<function <lambda> at 0x102f8e738>
```

λ Expressions: One More Example

A λ expression with no variables is analogous to a constant

```
>>> five = lambda : 5
>>> five
<function <lambda> at 0x102fa22f8>
>>> five()
5

>>> type(five)
<class 'function'>
>>> type(five())
<class 'int'>
```

λ Expressions: Another Example

When the built in sorting function of Python, `sorted`, is applied to a list of tuples, sorting is done according to the first (index 0) element in each tuple.

```
>>> lst=[(21,"Judas Iscariot"),(42,"Bartholomew"),
         (13,"Simon the Zealot")]
>>> sorted(lst)
[(13, 'Simon the Zealot'), (21, 'Judas Iscariot'),
(42, 'Bartholomew')]
```

Using a λ expression, we can easily instruct `sorted` to sort according to, say, the second (index 1) element in each tuple.

```
>>> sorted(lst,key=lambda x: x[1])
[(42, 'Bartholomew'), (21, 'Judas Iscariot'),
(13, 'Simon the Zealot')]
```

(In case you were wondering, these are three out of Jesus' Twelve Apostles, with Judas Iscariot being the most (in)famous one.)

And Now to Something Completely Different: Numeric Derivatives and Integrals

The sea was wet as wet could be,
The sands were dry as dry.
You could not see a cloud, because
No cloud was in the sky:
No birds were flying overhead –

There were no birds to fly.
The Walrus and the Carpenter
Were walking close at hand;
They wept like anything to see
Such quantities of sand:
"If this were only cleared away,"
They said, "it would be grand!"

Through the Looking-Glass and What Alice Found There: Lewis Carroll, 1871.

λ Expressions: A Few More 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, other than changing the default key in `sorted`?

The Derivative as a "High Order Operator"

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

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

- 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, this `diff` is a "high order function". However, for Python this is nothing unusual.

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

- On input `f`, a real value function, `diff(f)` returns another Python function, which is (a numeric approximation to) the derivative of `f`.

```
>>> diff(lambda x: x**3)  # derivative of f(x)=x**3
<function <lambda> at 0x300d978> # outcome of diff
```

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. We recommend you explicitly specify the original parameter name when setting such a different value (even though in the case of a single default parameter, this is not required by Python).

```
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)
```

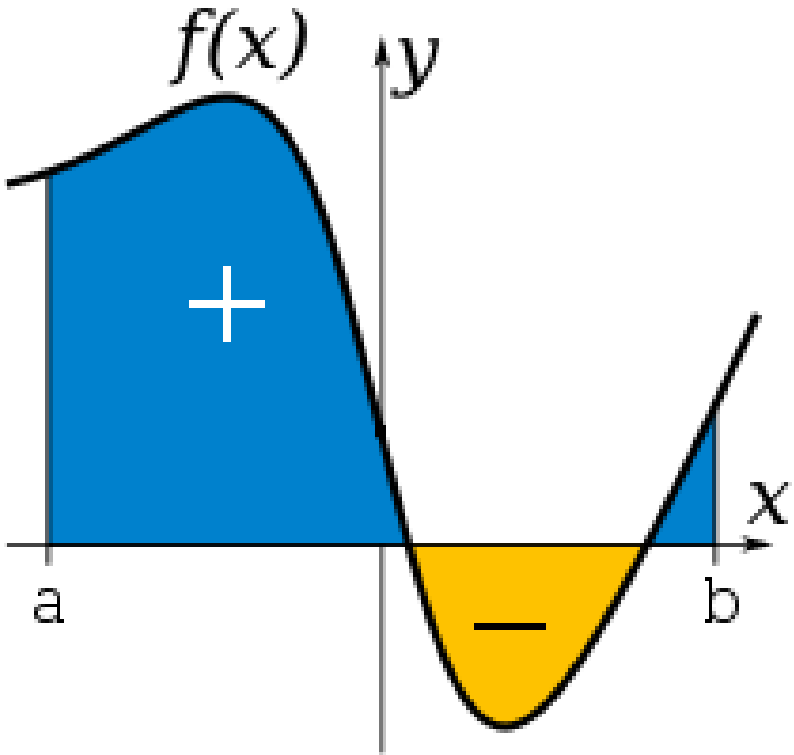
We can now apply this mechanism with different degrees of resolution.

```
>>> 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.

Definite Integrals

Let $f : \mathbb{R} \mapsto \mathbb{R}$ be a real valued function. Under some mild conditions (*e.g* piecewise continuity), its definite integral $\int_a^b f(x)dx$ between points a, b is defined as the signed area between the curve of $f(x)$ and the x-axis.

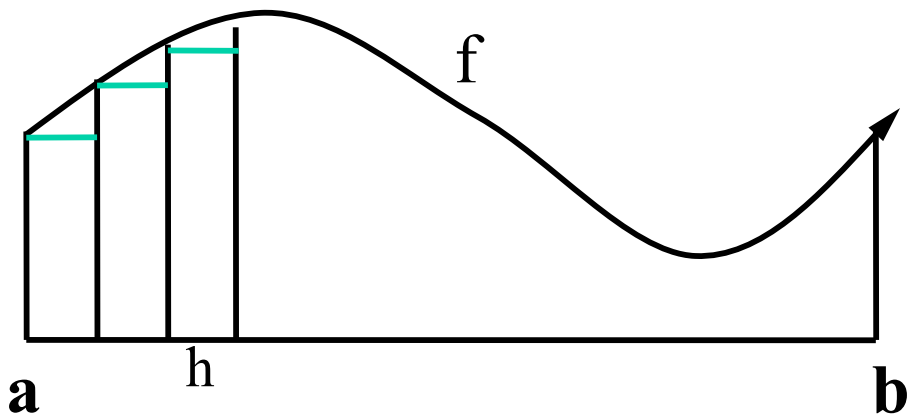


(drawing from Wikipedia.)

Computing the Integral

To approximate the definite integral, $\int_a^b f(x)dx$, we consider a (small) positive value h, and look at the finite arithmetic sequence of points, starting at a, going up to b, with "jumps" equal h,

$$a, a+h, a+2h, \ldots, a+ih, \ldots, a+\left\lfloor \frac{b-a}{h} \right\rfloor h \ .$$



For each such point, the area of the width h rectangle, bounded between the x-axis and the function $f(x)$ at point $a+ih$, equals $h \cdot f(a+ih)$.

Computing the Integral, cont.

Summing all these "rectangle terms", we get the expression

$$hf(a) + hf(a+h) + \ldots + hf(a+ih) + \ldots + hf\left(a + \left\lfloor \frac{b-a}{h} \right\rfloor h\right)$$

$$= h \sum_{i=0}^{\left\lfloor \frac{b-a}{h} \right\rfloor} f(a+ih)$$

These "Riemann sums" converge, as $h \to 0$, to the definite integral. As was the case with the derivative, we will not deal with the limit process, but instead take a small, albeit fixed, value for h.

Size of Intervals, h

Summing all these "rectangle terms", we get the expression

$$hf(a) + hf(a+h) + \ldots + hf(a+ih) + \ldots + hf\left(a + \left\lfloor \frac{b-a}{h} \right\rfloor h\right)$$

$$= h \sum_{i=0}^{\left\lfloor \frac{b-a}{h} \right\rfloor} f(a+ih)$$

The question of what values of h to take in order to guarantee that the resulting approximation is within a desired precision δ is

(a) dependent on properties of f (like the extreme values of its slope within the interval $[a, b]$)

(b) studied extensively within numerical mathematics.

The Integral as a "High Order Operator"

- Our input will be a real valued function, f.
- We want to return as the output another function, the definite integral, whose input is two endpoints a, b of an interval, and whose output is an approximation to the definite integral $\int_a^b f(x)dx$.
- The approximation will be computed by taking the Rieman sum with the parameter $h = 0.001$.
- Remarks: This parameter $h = 0.001$ can obviously be changed at will. Furthermore, the use of default parameter's value to control the diff, like we did with the derivative, is possible and even desirable. And we should verify that $a < b$, and further that $h \ll b - a$.

Python Code: The Integral as a "High Order Operator"

```
def integral(f,h=0.001):
    """ definite integral: function of a, b """
    return lambda a,b: \
           h*sum(f(a+i*h) for i in range(0,int((b-a)/h)))
# Recall that \ (backslash) denotes explicit line continuation

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

Running the code on a specific example

```
>>> integral(square)(0,1)
0.33283349999999995 # should be (1**3)/3=0.33333

>>> integral(diff(square))(0,2)
3.999999999996643
>>> integral(diff(square),h=0.000001)(0,2)    # higher resolution
4.001997999999674
>>> integral(diff(square))(0,100)
10000.00000002182
>>> integral(diff(square),h=0.000001)(0,100) # higher resolution
10000.0999000218  # much higher run time, LESS precision
>>>
```

Python Code: The Integral as a "High Order Operator"

Running the code on a specific example

```
>>> integral(diff(square))(0,2)
3.999999999996643
>>> integral(diff(square),h=0.000001)(0,2)    # higher resolution
4.00199799999674  # slightly LESS precision
>>> integral(diff(square))(0,100)
10000.00000002182
>>> integral(diff(square),h=0.000001)(0,100)  # higher resolution
10000.0999000218  # much higher run time, much LESS precision
>>>
```

We took the (numeric) derivative of square$(x) = x^2$, which is (or at least should be) $2x$. We then took the definite integral of this function over the intervals $[0, 2]$ and $[0, 100]$, respectively. The results are satisfyingly close to what they should be (4 and $10,000$, respectively).

Notice that increasing the resolution ($h = 0.000001$ instead of the default $h = 0.001$) actually decreased the outcome's precision.

Integration: Additional Examples

```
>>> diff_cos=diff(math.sin)
>>> integral(diff_cos)(0,math.pi/2)
0.999999682931 8186
>>> math.sin(math.pi/2)
1.0

>>> integral(diff_cos)(0,math.pi)
0.00059265355 1937746
>>> integral(diff_cos,h=0.0000001)(0,math.pi)
-0.0009998463 2690321
>>> math.sin(math.pi)
1.2246467991473532e-16
```

Again, we get good agreements (up to small numeric fluctuations) between the original function, $\sin(\cdot)$ and the integral of its derivative.

A smaller value h had caused more exhaustive computation of the integral, yet did not result in higher precision. This is in contrast to the phenomena regarding differentiation (why?).

Numeric vs. Symbolic Differentiation

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

>>> diff(penta)(2)
80.0800400099888     # 5*2**4=80
>>> diff_param(penta,h=0.0000001)(5)
3125.00134787115   # 5*5**4=3125
```

- You may argue that such an outcome is not what we are really after.
- penta$(x) = x^5$, so the derivative should produce penta$'(x) = 5x^4$.
- Likewise, we may want to get $\sin(x)' = \cos(x)$.
- The `diff` operator we defined is numerical. It transforms a "numeric" function into a different "numeric" function.
- It is very different from symbolic differentiation, alluded to above.

Symbolic Functions Representation (for reference only)

Let us define a list of "atomic functions":

- Constant functions $f(x) = c$
- The identity function $f(x) = x$
- The sine function $f(x) = \sin(x)$
- The cosine function $f(x) = \cos(x)$
- The exponentiation function $f(x) = e^x$
- The logarithm function $f(x) = \ln(x)$

Functions can be recursively combined using negation ($-f(x)$), addition ($f(x) + g(x)$), subtraction ($f(x) - g(x)$), multiplication ($f(x) \cdot g(x)$), division ($f(x)/g(x)$), exponentiation ($f(x)^{g(x)}$), and function composition ($f(g(x))$).

Symbolic Differentiation: Atomic Functions

The differentiation operator, which we will denote by diff, operates recursively on the function structure.
If f is one of the "atomic functions", its derivative is determined a priori, as following:

- diff(constant)$=0$,
- diff(identity)$=1$,
- diff$(\sin)=\cos$,
- diff$(\cos)=-\sin$,
- diff$(e^x) = e^x$,
- diff$(\ln(x)) = 1/x$.

Symbolic Differentiation: Compound Functions

For compound functions, we recursively apply the known differentiation rules

- $\mathrm{diff}(f \pm g) = \mathrm{diff}(f) \pm \mathrm{diff}(g)$,
- $\mathrm{diff}(f \cdot g) = \mathrm{diff}(f) \cdot g + f \cdot \mathrm{diff}(g)$,
- $\mathrm{diff}(f/g) = (\mathrm{diff}(f) \cdot g - f \cdot \mathrm{diff}(g))/g^2$,
- $\mathrm{diff}(f(g)) = (\mathrm{diff}(f)(g)) \cdot \mathrm{diff}(g)$ (chain rule).
- Remark: $f(x)^{g(x)} = e^{g(x)\ln[f(x)]}$, so $\mathrm{diff}(f^g)$ can be computed by the chain rule, $\mathrm{diff}(f^g) = e^{g\ln[f]} \cdot \left(\mathrm{diff}(g)\ln[f] + g(x)\frac{\mathrm{diff}(f)}{f}\right)$.

Symbolic Differentiation: Compound Functions

Symbolic differentiation thus becomes a question of manipulating strings. There are additional issues tackled in any Computer Algebra package (*e.g.* Sage, Maple, or Mathematica), such as symbolic integration, polynomial (not integer) factorization, or expression simplification (*e.g.* replacing $x + x^2 + x + 2 \cdot x^2$ by $2 \cdot x + 3 \cdot x^2$). Many of these issues involve deep algorithmic and mathematical understanding.

We will not develop code for symbolic differentiation, or any of the other tasks. We may get back to this issue in the future.

But for now, we'll just show a screenshot from Sage.
Sage is a free, open-source, symbolic mathematics software system.

Incidentally, it has a Python based interface.

Symbolic Mathematics in Sage: Sample screenshot

```
f(x)=x^5 - 6*x^3 - 12*x^2 + 5*x + 60
factor(f(x))
```

```
(x^2 - 5)*(x^3 - x - 12)
```

```
diff(f(x))
```

```
5*x^4 - 18*x^2 - 24*x + 5
```

```
integral(_,x)
```

```
x^5 - 6*x^3 - 12*x^2 + 5*x
```

```
H(x)=-(x*log(x)+(1-x)*log(1-x))  # the binary entropy function
diff(H(x),x)
```

```
log(-x + 1) - log(x)
```

Sage (and other symbolic mathematics packages, like Maple, Mathematica, etc.) are certainly capable of much more. These capabilities are highly useful in diverse contexts.

And Now to Something Completely Different: Floating Point Arithmetic

"If seven maids with seven mops
Swept it for half a year.
Do you suppose," the Walrus said,
"That they could get it clear?"
"I doubt it," said the Carpenter,
And shed a bitter tear.

"O Oysters, come and walk with us!"
The Walrus did beseech.
"A pleasant walk, a pleasant talk,
Along the briny beach:
We cannot do with more than four,
To give a hand to each."

Through the Looking-Glass and What Alice Found There: Lewis Carroll, 1871.

Floating Point Arithmetic

Our real number arithmetic is done using floating points numbers, of bounded accuracy (which is determined by the "physical word size", the operating system you are using, the version of the interpreter, etc., etc.).

Floating point arithmetic carries many surprises for the unwary (and the infidels). This follows from the fact that floating numbers are represented as a number in binary, namely the sum of a fixed number of (positive and negative) powers of two.

The bad news is that very simple rational numbers cannot be represented this way. For example, the decimal $0.1 = 1/10$ cannot be represented as a sum of powers of two, since the denominator has prime factors other than 2, in this case, 5.

Wonders of Floating Point Arithmetic

A confusing issue is that when we type 0.1 (or, equivalently, 1/10) to the interpreter, the reply is 0.1.

```
>>> 1/10
0.1
```

This does not mean that 0.1 is represented exactly as a floating point number. It just means that Python's designers have built the `display` function to act this way. In fact the inner representation of 0.1 on most machines today is 3602879701896396 / 2**55.

```
>>> 1/10 == 3602879701896397 / 2**55
True
```

And so is the inner representation of 0.10000000000000001. Since the two have the same inner representation, no wonder that `display` treats them the same

```
>>> 0.10000000000000001
0.1
```

More Wonders of Floating Point Arithmetic

Once we realize this inherent restriction of floating point arithmetic, some mysteries are resolved. For example

```
>>> 0.1+0.1 == 0.2
True
>>> 0.1+0.1+0.1 == 0.3
False
```

And indeed,

```
>>> 0.1+0.1
0.2
>>> 0.1+0.1+0.1
0.30000000000000004
```

And Even More Wonders of Floating Point Arithmetic

As we have pointed out, floating point precision is determined by the "physical word size", the operating system you are using, the version of the interpreter, etc. etc. For example, on my home computer (MacBook Pro, running MAC OSX version 10.6.8, Build 10K549, using Python version 3.2.2 and the IDLE interpreter), I got the following

```
>>> type(10**(-324))
<class 'float'>
>>> 10**(-324) == 0.
True
>>> 10**(-323) == 0.
False
```

So supposedly precision on that specific machine around zero is 323 digital points. This may seem incompatible with having

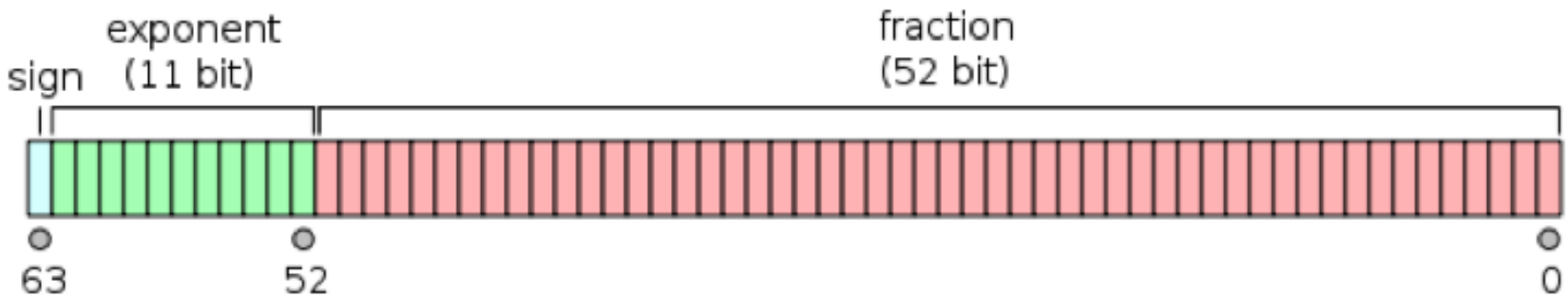
```
>>> 0.10000000000000001 == 0.1
True
```

Representation of Floating Point Numbers (reference only)

Suppose we deal with a machine with 64 bit words. A floating point number is typically represented by

$\texttt{sign} \cdot 2^{\texttt{exponent}-2013} \cdot (1 + \texttt{fraction})$.

- The `sign` is ± 1 (1 indicates negative, 0 indicates non-negative).
- The `exponent` is an 11 bits unsigned integer, so it represents an integer between 0 and $2^{11} - 1 = 2047$.
  So $-2013 \leq \texttt{exponent} - 1023 \leq 2014$.
- The `fraction` is a sum of negative powers of 2, represented by 52 bits, $0 \leq \texttt{fraction} \leq 1 - 2^{-52}$.

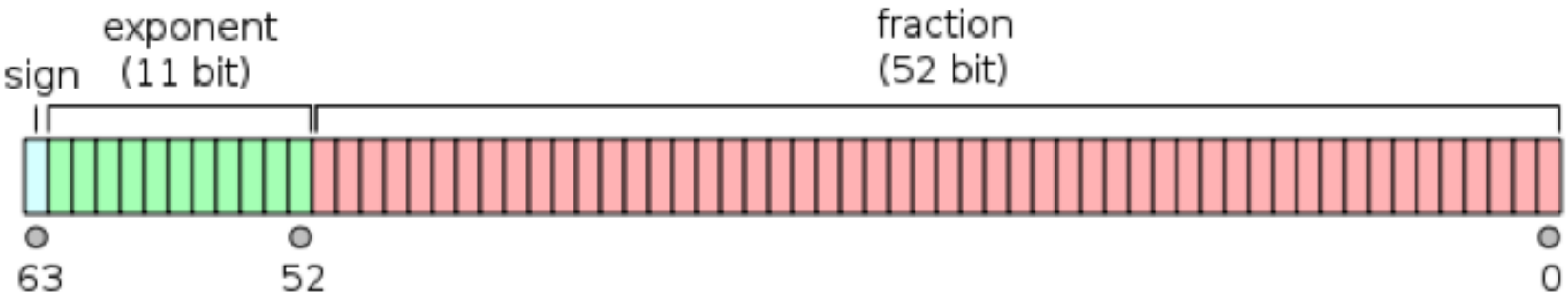


(figure from Wikipedia)

In particular, the largest floating point number in a 64 bit word is smaller than 2^{1024}, and larger than 2^{1023}.

Representation of Floating Point Numbers

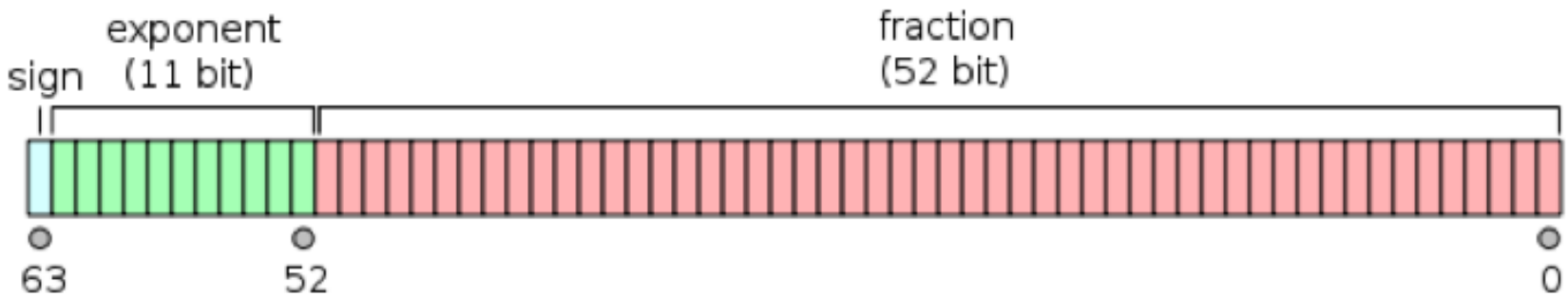
```
>>> 2.**1024
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    2.**1024
OverflowError: (34, 'Result too large')
>>> 2.**1023
8.98846567431158e+307
```



In particular, the largest floating point number in a 64 bit machine is smaller than 2^{1024}, and larger than 2^{1023}. We can try looking for this maximum,

```
>>> sum (2.0**i for i in range(971,1024))
1.7976931348623157e+308
>>> sum (2.0**i for i in range(970,1024))
inf
```

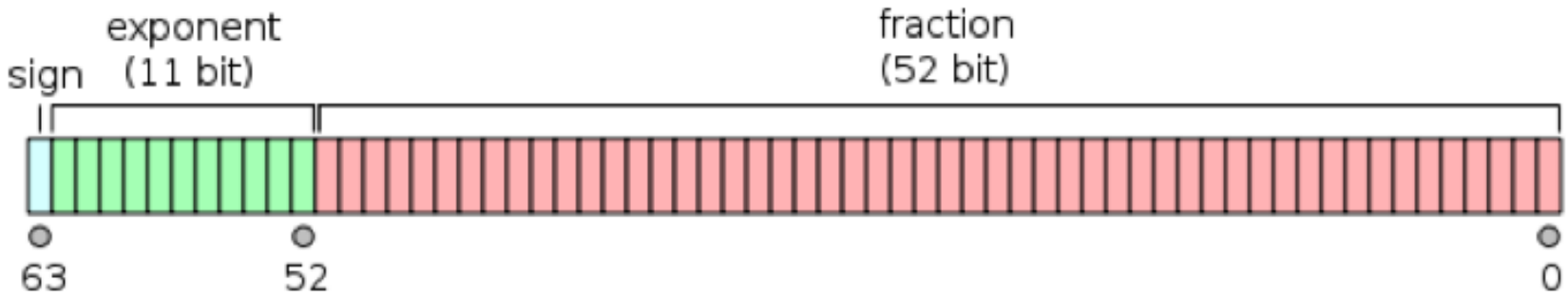
Representation of Floating Point Numbers: Special Case(s)



`0.0` is represented as
`sign=0`, `exponent=` 11 zeroes, and `fraction=` 52 zeroes.

`-0.0` is represented as
`sign=1`, `exponent_plus=` 11 zeroes, and `fraction=` 52 zeroes.

Representation of Floating Point Numbers (for reference only)



We can also probe Python for exact values of various floating points constants (and even understand most of them).

```
>>> sys.float_info
sys.floatinfo(max=1.7976931348623157e+308,
max_exp=1024,
max_10_exp=308,
min=2.2250738585072014e-308,
min_exp=-1021,
min_10_exp=-307,
dig=15,
mant_dig=53,
epsilon=2.220446049250313e-16,
radix=2,
rounds=1)
```

Hackers' Delight: Representation of Floating Point Numbers, the Full Monty (for reference only)

The following (rather obscure) code displays all 64 bits of a floating point number. It is brought to you as a courtesy of the mythological Python's guru in the course staff (hint: not your lecturer):

```
import  struct

def display_float(x):
    """ prints 64 bits in the representation of the float x """
    if isinstance(x,float):
        q, = struct.unpack('Q', struct.pack("d", x)) # struct magic
        full_bin="{:064b}".format(q)
        sign= full_bin[0]
        exponent_plus=full_bin[1:12]
        fraction=full_bin[12:]
        return sign+" "+exponent_plus+" "+fraction
    else:
        return None
```

The true exponent is determined by `exponent_plus - 1023`. This enables us to get both positive and negative values.

The Full Monty: Specific examples (for reference only)

```
>>> display_float(1.)
'0 01111111111
      0000000000000000000000000000000000000000000000000000'

>>> display_float(1/2)
'0 01111111110
      0000000000000000000000000000000000000000000000000000'

>>> display_float(2.0)
'0 10000000000
      0000000000000000000000000000000000000000000000000000'

>>> display_float(0.3)
'0 01111111101
      0011001100110011001100110011001100110011001100110011'
```

For example, the `exponent` for 1.0 in binary is `01111111111`, and `int("01111111111",2)` is 1023.
So $\texttt{sign} \cdot 2^{\texttt{exponent}-1023} \cdot (1 + \texttt{fraction})$ equals $1 \cdot 2^0 \cdot (1 + 0) = 1$.

The Full Monty: Specific examples, cont. (reference only)

```
>>> display_float(0.0)
'0 00000000000
      0000000000000000000000000000000000000000000000000000'

>>> display_float(-0.0)
'1 00000000000
      0000000000000000000000000000000000000000000000000000'
```

Arithmetic of Floating Point Numbers

Multiplication is done by multiplying the two fractions, and adding the two exponents.

Addition is done by first representing both numbers with the same exponent, then adding the weighted fractions, then converting back so that the fraction is smaller than 1.

Subtraction and division are analogous to addition and multiplication, correspondingly.

Addition and subtraction often lead to substantial loss of precision.

```
>>> a=1111111111111111111111111111111111111111111111111.0
>>> b=1111111111111111111111111111111.0
>>> a/b
1.0000000000000001e+18
>>> a-b == a
True
```

You may encountered such behavior while solving a future assignment.

Relaxed Equality for Floating Point Arithmetic

We may attempt to solve some non intuitive issues with floating point numbers by redefining equality to mean equality up to some epsilon. For example

```
def float_eq(num1,num2,epsilon=10**(-10)):
    """ re-defines equality of floating point numbers """
    assert(isinstance(num1,float) and isinstance(num2,float))
    return abs(num1-num2)<epsilon
```

This indeed solves one problem

```
>>> float_eq(0.1+0.1+0.1,0.3)
True
```

But now this new equality relation is not transitive

```
>>> float_eq(0.,2*11**(-10))
True
>>> float_eq(2*11**(-10),4*11**(-10))
True
>>> float_eq(0.,4*11**(-10))
False
```

Floating Point Arithmetic vs. Rational Arithmetic

So, no matter how we turn it around, floating point arithmetic introduces some challenges and problems we are not very used to in "everyday mathematical life".

Questions:

- Can we do something about this?

Answers:

1. Yes, we can! (Unbounded precision, rational arithmetic):

```
>>> from fractions import Fraction
>>> Fraction (6,10)  # unbounded precision 6 divided by 10
Fraction(3, 5)
>>> Fraction (6,10) + Fraction(7,12)
Fraction(71, 60)
>>> Fraction (1,10) +  Fraction (1,10) + Fraction (1,10)
Fraction(3, 10)
```

2. We may also write our own Rational class and implement such unbounded precision arithmetic ourselves (not that hard).

Floating Point Arithmetic vs. Rational Arithmetic

So, no matter how we turn it around, floating point arithmetic introduces some challenges and problems we are not very used to in "everyday mathematical life".

Questions:

- Should we bother?

Answers:

3. Usually not: Typically the results of rational (unbounded precision) arithmetic for numerical computations are very similar to results from floating point arithmetic. Yet rational arithmetic is not for free – huge numerators and denominators tend to form, slowing down computation significantly, for no good reason.
4. Still, in some singular (and fairly rare) cases, numerical computations can be unstable, and the outcomes of floating point vs. rational arithmetic can be very different.

And Now to Something Completely Different: Finding Roots of Real Valued Functions

The eldest Oyster looked at him,
But never a word he said:
The eldest Oyster winked his eye,
And shook his heavy head–
Meaning to say he did not choose
To leave the oyster-bed.

But four young Oysters hurried up,
All eager for the treat:
Their coats were brushed, their faces washed,
Their shoes were clean and neat–
And this was odd, because, you know,
They hadn't any feet.

Through the Looking-Glass and What Alice Found There: Lewis Carroll, 1871.

Finding Roots of a Real Valued Function, Take 1 [†]

You are given a black box that computes a real valued function, $f(x)$.

You are asked to find a root of $f(x)$ (namely a value a such that $f(a) == 0$ or at least $|f(a)| < \varepsilon$ for a small enough ε).

What can you do?

Not much, I'm afraid. Just go over points in some arbitrary/random order, and hope to hit a root.

[†] thanks to Prof. Sivan Toledo for helpful suggestions and discussions related to this part of the lecture

Finding Roots of Real Valued Function, Take 2

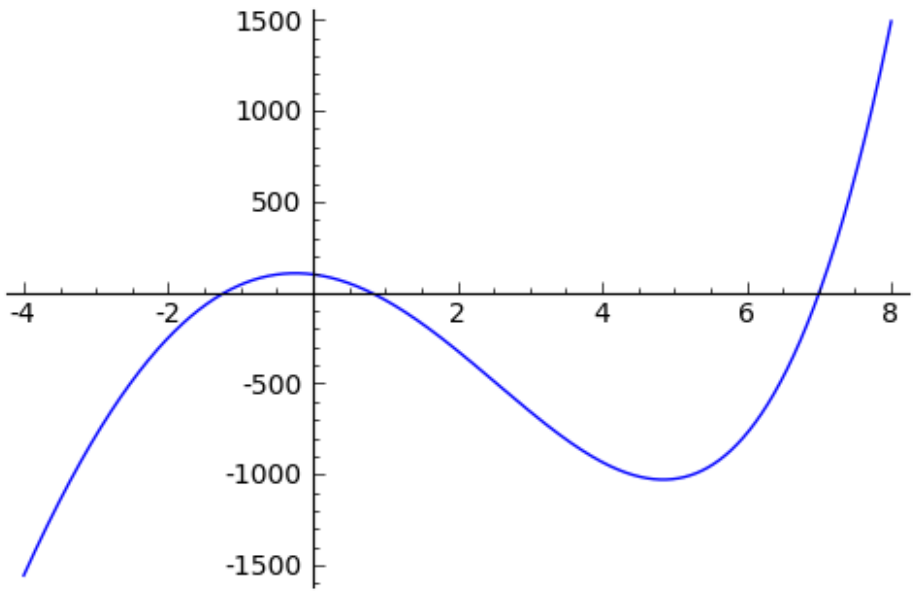
You are given a black box to compute the real valued function $f(x)$. On top of this, you are told that $f(x)$ is continuous, and you are given two values, L and U, such that $f(L) < 0 < f(U)$.

You are asked to find a root of $f(x)$ (namely a value a such that $f(a) == 0$ or at least $|f(a)| < \varepsilon$ for a small enough ε).

What can you do?

The Intermediate Value Theorem

Suppose that $f(x)$ is a continuos real valued function, and $f(L) < 0 < f(U)$ (where $L < U$, and both are reals). The intermediate value theorem (first year calculus) claims the existence of an intermediate value, C, $L < C < U$, such that $f(C) = 0$. There could be more than one such root, but the theorem guarantees that at least one exists.



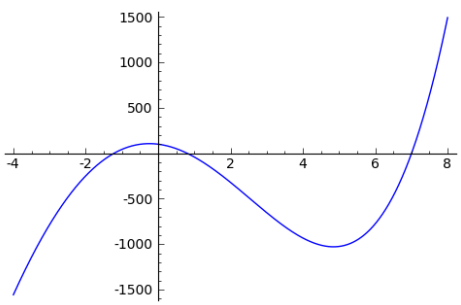
For example, in this figure, $f(-4) < 0 < f(8)$, so there is a C, $-4 < C < 8$, such that $f(C) = 0$ (in fact there are three such C .

Root Finding Using Binary Search

Suppose that $f(x)$ is a continuos real valued function, and $f(L) < 0 < f(U)$ (where $L < U$). Compute $M = f((L+U)/2)$.

- If $M = 0$, then M is a root of $f(x)$.
- If $M < 0$, then by the intermediate value theorem, there is a root of $f(x)$ in the open interval $((L+U)/2, U)$.
- If $M > 0$, then by the intermediate value theorem, there is a root of $f(x)$ in the open interval $(L, (L+U)/2)$.

Looks familiar?



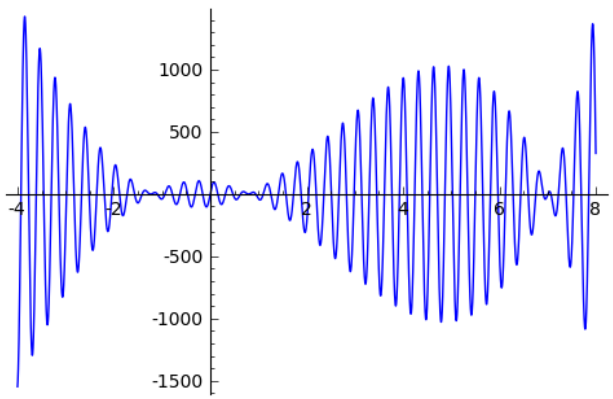
By performing binary search on the interval, we converge to a root of $f(x)$ (will stop when $|M| < \varepsilon$).

Root Finding Using Binary Search: Potential Problems

The intermediate value theorem deals with real numbers. Our computers approximate them using floating point numbers. Unlike reals, floating point numbers have limited precision.

If the function under consideration is rather tame, there should be no problem.

But wild functions can cause overflow (values too large to be represented by floats) or underflow (non zero numbers represented as zero), and search can thus miss roots even if the roots by themselves are not problematic (*i.e.* can be represented well with floats).



We may see such example (in a recitation and/or a future homework).

Finding Roots of Real Valued Function, Take 3

You are given a black box to compute the real valued function $f(x)$. On top of this, you are told that $f(x)$ is differentiable (is smooth enough to have a derivative).

You are asked to find a root of $f(x)$ (namely a value a such that $f(a) == 0$ or at least $|f(a)| < \varepsilon$ for a small enough ε).

What can you do?
(here, we'll start discussing the Newton-Raphson iteration.)

An Example: Computing Square Roots

- Suppose we want to find a square root of 1732.
- $41^2 = 1681$ and $42^2 = 1764$.
- Since $1681 < 1732 < 1764$, we can safely claim that 1732 is not a square of any integer.
- What next? We'll start with some initial guess, denoted x_0, and design a sequence $x_1, x_2, \ldots$ that will converge to $\sqrt{1732}$.
- Define $x_{n+1} = x_n - \frac{x_n^2 - 1732}{2x_n}$.
- Remark: Right now, this expression sort of comes out of the blue.

Computing Square Root of 1732: Code

```
def iter1732(x):     # compute next sequence element and its square
    return x-(x**2-1732)/(2*x),(x-(x**2-1732)/(2*x))**2
```

Let us pretend we got no idea where $\sqrt{1732}$ is, start with $x_0 = 10$, and iterate the function to get $x_1, x_2, \ldots$, as well as their squares.

```
>>> iter1732(10)
(91.6, 8390.56)     # x1, x1**2
>>> iter1732(_[0])  # apply iter1732 to first element
(55.25414847161571 4, 3053.020923323353)  # x2, x2**2
>>> iter1732(_[0])
(43.30010556385136, 1874.8991418406715)  # x3, x3**2
>>> iter1732(_[0])
(41.65000402275986, 1734.7228350959124)  # x4, x4**2
>>> iter1732(_[0])
(41.61731692992759, 1732.0010684460376)  # x5, x5**2
>>> iter1732(_[0])
(41.6173040933716, 1732.0000000001646)  # x6, x6**2
>>> iter1732(_[0])
(41.617304093369626, 1732.0000000000002)  # x7, x7**2
>>> iter1732(_[0])
(41.617304093369626, 1732.0000000000002)  # convergence, x7==x8
```

Another Example: Computing Square Roots

- Suppose we want to find a square root of 9387.
- $96^2 = 9216$ and $97^2 = 9409$.
- Since $9216 < 9387 < 9409$, we can safely claim that 9387 is not a square of any integer.
- What next? We'll start with some initial guess, denoted x_0, and design a sequence $x_1, x_2, \ldots$ that will converge to $\sqrt{9387}$.
- Define $x_{n+1} = x_n - \frac{x_n^2 - 9387}{2x_n}$.
- This expression still comes out of the blue.
- But please note that if for some obscure reason this sequence converges to a non-zero limit, and we denote this limit by x_∞, we'll have $x_\infty^2 - 9387 = 0$. In other words, x_∞ will be a square root of 9387.

Computing Square Root of 9387: Code

```
def iter9837(x):     # compute next sequence element and its square
    return x-(x**2-9387)/(2*x),(x-(x**2-9387)/(2*x))**2
```

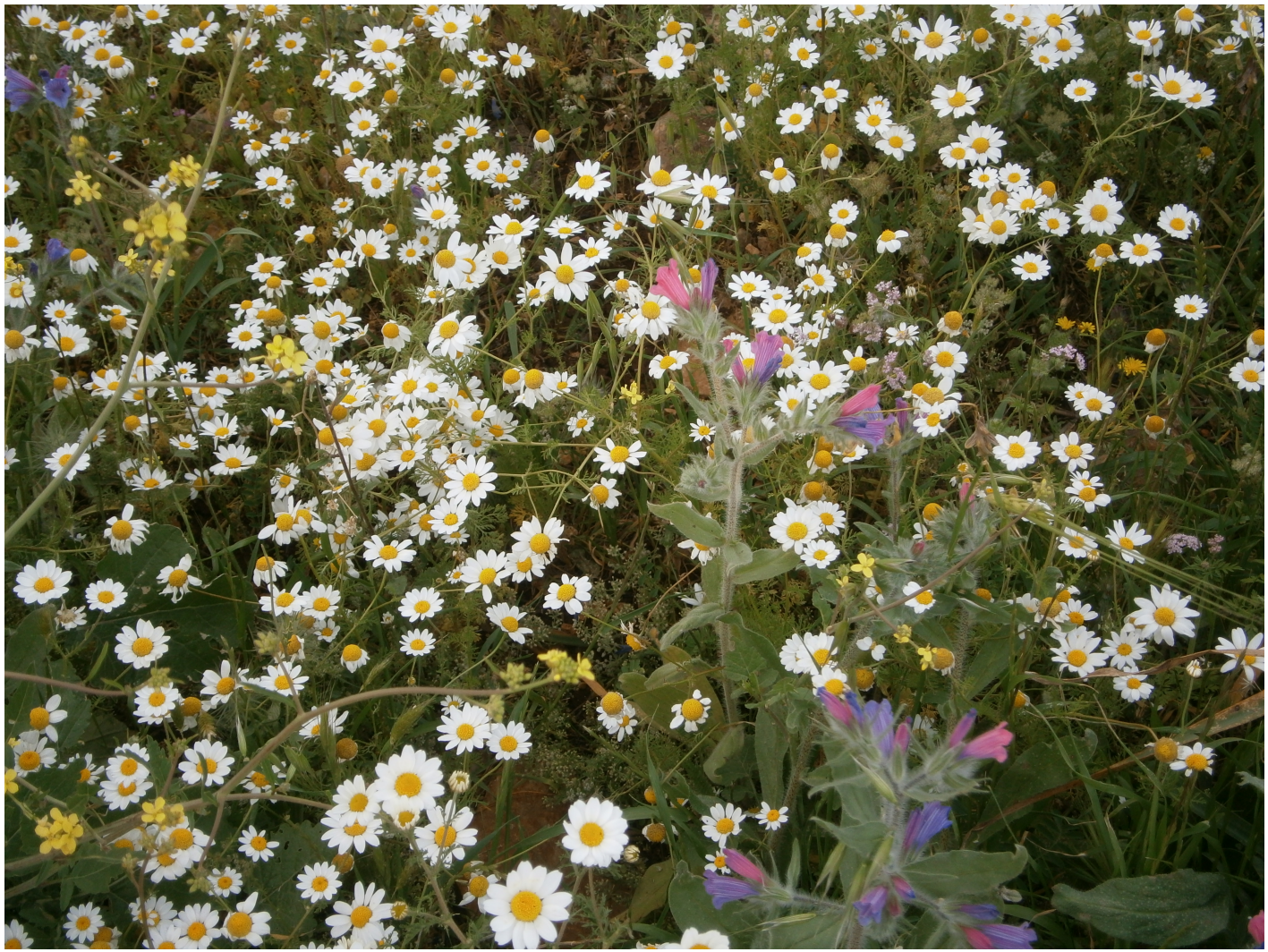
Let us pretend we got no idea where $\sqrt{9387}$ is, start with $x_0 = 13.7$ this time, and iterate the function to get $x_1, x_2, \ldots$, and their squares.

```
>>> iter9837(13.7)
(349.4412408759124, 122109.18082489743) # x1, x1**2
>>> iter9837(_[0]) # apply iter9837 to first element
(188.15206312696233, 35401.19885893242)  # x2, x2**2
>>> iter9837(_[0])
(119.02128022032366, 14166.065145284809) # x3, x3**2
>>> iter9837(_[0])
(98.94476475839053, 9790.066473093242) #x4, x4**2
>>> iter9837(_[0])
(96.90793909066838, 9391.148658800692)  # x5, x5**2
>>> iter9837(_[0])
(96.88653393625265, 9387.000458180635)  # x6, x6**2
>>> iter9837(_[0])
(96.88653157173088, 9387.000000000005) # x7, x7**2
>>> iter9837(_[0])
(96.88653157173086, 9387.0) # x8, x8**2
>>> iter9837(_[0])
(96.88653157173086, 9387.0) # convergence, x8==x9
```

This All Leads to Newton Raphson Iteration

Which we will discuss in next class, *i.e.* on Monday.

Last, But Not Least



חג אביב שמח

(וכשר לאוכלי קטניות)