Part III

Python Exercises

32 Foreword

The sections that follow are your first exercises, and your entry point into this Python class. Alternate between answering the questions below and reading the lecture notes in Part II[p45].

Whenever I direct you to a specific section, make sure to read it carefully, and to seek out, in the lecture notes, whatever information you may be missing to understand those sections.

Note that you will be expected to have read and more or less understood all of Part II[p45] by the end of the semester. Points that are a bit beyond the scope of this class and will not be tested in the exam (such as anything beyond the basics of Object Oriented Programming) are indicated as such in the notes; you still need to read those parts, though.

We have eleven **lab classes** (three of which are in autonomy) and two **lectures**, roughly at the $\frac{1}{3}$ and $\frac{2}{3}$ marks.

During the lectures, I shall discuss the solution to as many exercises as time allows, and take your questions.

The new key insights and core competencies targeted by each question or exercise are written in this format. They should become clear *after* having solved the questions and having discussed them with me.

32.1 An open letter to Python Gods

A note to those who already know Python. Or *think* they do.

You may skip this diatribe if you *don't* think that.

Even if you are already well practised in Python, please *do not rush through the exercises at all speed*; do not skip them to get to something more "interesting" more quickly. It is quite unlikely that you already know *everything* in Part II[p45], and there is a difference between **(1)** having enough tools to be able to cobble together a purported solution to a problem, and **(2)** using the right tools to quickly produce short, efficient, reliable, and well-tested code that fully satisfies its specification. Those exercises, no matter how trivial some may seem to you, are opportunities for me to engage with you on those topics, check for bad habits you may have acquired, etc; do not neglect them.

Also note that, during the exams, I often ask questions testing your knowledge of specific Python idioms and structures, such as comprehension expressions, the limitations of sets and dictionaries, the specificity of Python's `int` type compared to that of other languages, etcetera. I had some cases in previous years of students entering the class fully confident that they already knew Python, because they had successfully completed some project in it. Thus, they paid no attention and did not put any effort into this class, and learned nothing new. They were then positively outraged to receive a failing grade in the final exam; dismayed that writing Java or C roughly translated into Python's syntax did not satisfy. Do not be them.

33 Basic data types, expressions, and functions

33.1 Conversion Celsius ↔ Fahrenheit

def <function>; return <expression>; assert for defensive programming

La formule de conversion entre ces deux unités étant

$$F = \frac{9}{5}C + 32\,,$$

nommer, écrire et documenter les deux fonctions de conversion `F_to_C` et `C_to_F` pour passer d'une unité à l'autre.

Quelles sont les conditions d'utilisation de ces fonctions ? As a reminder, the absolute zero is $-459.67°F$ and $-273.15°C$.

On veillera à utiliser des assertions (cf. Sec. 21.6.6[p101]: "Assertions: cheap unit testing and preconditions enforcing") afin de tester ces conditions d'utilisation. Please read that section very carefully. In particular, do not burden your assertions with redundant error messages.

Notons que, bien que le type des arguments d'entrées fasse moralement partie des préconditions de toute fonction, on ne demande pas de le vérifier programmatiquement dans ces TDs, et il est peu idiomatique de le faire en Python.

Do not use `input()`! That is what the function's arguments are for. If I ever want user interaction, I shall ask for it explicitly. Use **`print`**s and, whenever possible, **`assert`**s in your code to test the functions on different values.

Do not confuse **`return`** and **`print`**: whenever I ask for a function, it must **`return`** something, so that I can use the function in later computation, not **`print`** it.

33.2 Floating-point comparison: almost there

float has finite precision; bool/predicates are very simple;
assert for cheap unit tests

Let us check that our two conversion functions are coherent with one another, by testing that their absolute zeros match.

However, we cannot simply test equality between floating point numbers, for reasons discussed in Sec. 21.2[p76]: "Floating-point numbers: **`float`**": there *may* be a loss of precision:

```
>>> F_to_C(-459.67) == -273.15
False

>>> F_to_C(-459.67)
-273.15000000000003
```

Although you may or may not observe it depending on the exact way you performed the computation:

```
>>> [ (a,b)
      for F in range(20)
      for a,b in [[5/9*(F-32), 5*(F-32)/9]]
      if a!=b ]

---------------------------------------------

[(-15.555555555555557, -15.555555555555555),
 (-12.222222222222223, -12.222222222222221),
 (-11.666666666666668, -11.666666666666666),
 ( -7.777777777777779,  -7.777777777777778)]
```

Instead of running that risk, we shall test whether the two values are *very, very close*.

(21) Define a predicate `isalmost(n,m,d=1e-13)` that tests whether n and m are at a distance at most d [ad].

(22) Verify that the following assertions are satisfied:

```
assert isalmost ( F_to_C(-459.67) , -273.15 )
assert isalmost ( C_to_F(-273.15) , -459.67 )
assert all( isalmost( efc:=F_to_C(C_to_F(c)), ec:=c )
       and  isalmost( efc:=C_to_F(F_to_C(c)), c )
       for c in range(-273, 200) ), (ec, efc)
```

(Almost) always leave your assertions in your code, to prevent future regressions.

33.3 Taking root

33.3.1 Greatest root

reading a specification and enforcing valid inputs with `assert`; `None` as `null`

Écrire et documenter une fonction `greatest_root` telle que `greatest_root(a,b,c)` retourne la plus grande racine réelle du polynôme de second degré $ax^2 + bx + c$ si elle existe, et `None` sinon. See Sec. 21.5[p93]: “Nihilism: `NoneType`: expression versus statement”.

[ad] Such a basic tool to manipulate floating-point numbers is of course provided in the standard library. This function is a simpler re-implementation of math.isclose.

Quelles sont les conditions d'utilisation (ou *préconditions*) de cette fonction ?

Indication: do all tuples $(a, b, c) \in \mathbb{R}^3$ describe a valid polynomial of the second degree?

Note: as mentioned in Sec. 21[p73]: "Basic data types", in Python, it is not idiomatic to test the type of parameters explicitly. While it is true that testing whether a, b, c are numerical values would be pertinent , let's not do that.

Once you have worked out what the preconditions are, make sure to enforce them through an **assert**.

Quick reminder from high school: the roots of $ax^2 + bx + c$ are given by

$$\frac{-b \pm \sqrt{\Delta}}{2a},$$

where $\Delta = b^2 - 4ac$ is called the *discriminant*. They are real if $\Delta \geqslant 0$.

The following assertions must be satisfied:

```
assert greatest_root(1,1,1) == None
assert greatest_root(1,-2,1) == 1
assert greatest_root(4,-4,-24) == 3
# we can do direct comparison because we know the results are exact,
# at least in this case, and we only want to prevent regressions
# This is overspecification, and might refuse correct versions of
# the function. However, we can deal with those problems, using
# isalmost, as they arise.
```

Feel free to add some more.

33.3.2 Real roots

carefully reading and respecting the specification
virtues of homogeneous return types and containers
unit tests on ranges of value to enforce consistency of implementations

Write a function `roots(a,b,c)` returning a tuple containing the real roots of the second-degree polynomial $ax^2 + bx + c$, in no particular order.

Note how this specification is formulated: you must *always* return a tuple. Do not return a tuple sometimes and None some other times.

The following assertions must be satisfied:

```
assert roots(1,1,1) == ()

assert roots(1,-2,1) in [ (1,1), (1,) ]
# I didn't specify whether single roots should be repeated,
# so both versions are valid

assert set(roots(4,-4,-24)) == {-2, 3}
```

```
# I did not specify the order of roots, hence the set test
```

Write an assertion testing, for all valid values of $a, b, c \in [\![-5, 5]\!]$, that the outputs of `greatest_root` and `roots` are coherent.

This can be done in one or two logical lines if you can have read Sec. 23.5[p148]: "Comprehension expressions" and Sec. 22.2[p107]: "Conditional expression: `.. if .. else ..` ternary operator", but I do not *require* that at this point.

34 Dungeons and magic methods

`ex_d20.tex` **WORK IN PROGRESS !**

`https://www.d20srd.org/srd/theBasics.htm#dice`

show no function `d`; implement class

Sec. 26.7[p186]: "String representations `str` and `repr`"

Sec. 26.10[p191]: "Special, magic, dunder methods" `__int__` does int implem cause auto-convert in arithmetic expressions? Nope.

Sec. 23.5[p148]: "Comprehension expressions" and sum optional

Counter for stats

Remark that it's magic under linspace plots later on. Hijack expressions into something more abstract.

35 We all `float` down here!

It is nice that we can solve the zeroes of a second-degree polynomial. Now let us do the same thing for the zeroes of *any* continuous function f. We shall use approximate numerical methods.

Keep in mind that floating point numbers are dangerous; don't turn your back on them. Be sure to read Sec. 21.2[p76]: "Floating-point numbers: `float`" in that regard.

35.1 The Zero Dichotomy

thinking recursively
less trivial effects of `float` precision loss
beware false simplicity of `int`
hide recursion from user with recursive subfunctions
why inclusive/exclusive ranges $[\![a, b[\![$ are cool

Recall the classical theorem of intermediate values:

Theorem 1 (Intermediate Values)**.** *If a function* $f : \mathbb{R} \to \mathbb{R}$ *is defined and continuous on a real interval* I*, then its image* $f(I)$ *is also an interval.*

It has an important corollary:

Corollary 2 (Bolzano's theorem – BT)**.** $\forall a, b \in I$, if $f(a)f(b) \leqslant 0$, then $\exists z \in [a, b] : f(z) = 0$.

Proof. $f([a, b])$ is an interval; it contains $f(a)$ and $f(b)$. Therefore it contains $[f(a), f(b)]$ (or $[f(b), f(a)]$). We have $f(a)f(b) \leqslant 0$, which means that if either $f(a)$ or $f(b)$ is positive, then the other must be negative, and vice versa. Thus we have $0 \in [f(b), f(a)] \subseteq f([a, b])$.

In short: f changes sign between a and b, so its curve must cross the abscissa. □

We are going to use Bolzano's theorem to implement a **dichotomic search** — or more precisely a binary search; also known as the bisection method — for a zero; that is to say, a value z such that $f(z) = 0$. What is a dichotomic search, you ask? That is what you do when you search for a word in the dictionary — I mean a *paper* dictionary, not an online one. (If you are not old enough to have manipulated one of those, use your imagination).

The principle is simple: you open the dictionary at some place, roughly down the middle (maybe you have better guesses if the word begins by A or Z, but it matters little in the end), and you determine, using the alphabetical order, whether the word you are looking for is to the left, or to the right, or your current position. Therein lies the "dichotomy", the "bisection": you have two mutually exclusive options: left and right. If you cut exactly in the middle each time, it's a binary search.

Then you repeat that process, with either the left or right part of the dictionary, which is of course much smaller, again and again until you are close enough to your target.

This is a very efficient process: $O(\log_2 N)$ where N is the size of the dictionary — or more generally of any sorted list. Not convinced of the logarithm? if the dictionary is twice as big, you open it at the middle, and choose left or right, and you're back to the original size. One more step to handle twice the size.

Let us do that to search for zeroes: start with a and b such that f changes sign between them, cut down the middle of $[a, b]$, and then there must be a zero either left or right; choose by testing on which side there is a sign change, and repeat until you're close enough to your taste. That is to say, until $|a - b|$ is smaller than your desired precision.

(23) **TODO VH:** Separate into two questions: normal precision and max, with assertions for each.

Write a function `di(f,a,b,d=1e-16)`, where $f : [a, b] \to \mathbb{R}$ is continuous and changes sign on $[a, b]$, that returns an approximation of a zero z, ideally within a precision d. You are not required to test whether the inputs satisfy the assumptions.

Tip: *there are two ways to write a dichotomy: recursively, and with a* **`while`** *loop. Write it recursively first, it's simpler. You'll write it with a* **`while`** *in the next question.*

As an example of how it must behave, let us approximate $\sqrt{2}$ as the positive root of $X^2 - 2$:

```
from math import sqrt

def g(x): return x**2 - 2

>>> res = di(g,1,2)
1.414213562373095
>>> sqrt(2)
1.4142135623730951
>>> sqrt(2)-res
2.220446049250313e-16
```

Printing each step of the process with `print(a,m,b, b-a)`, where `m` is the middle, we have:

```
1 1.5 2 1
1 1.25 1.5 0.5
1.25 1.375 1.5 0.25
1.375 1.4375 1.5 0.125
....
1.414213562373095 1.4142135623730951 1.4142135623730954
                                            4.440892098500626e-16
1.414213562373095 1.414213562373095 1.4142135623730951
                                            2.220446049250313e-16
```

This also illustrate why I said *ideally* within a precision d. Observe that our precision objective is not actually met in the example. Indeed I could run with `d=1e-160` and get the very same result.

This is because we are dealing with floating-point numbers, and thus loss of precision. It may well be that the middle becomes impossible to distinguish from a or b — because of loss of precision — before $|a - b|$ becomes quite small enough. In that case we run the risk of entering an infinite loop, so we must return the result we have now. That is precisely what happened here: a and m are the same in the last line, so our current approximation is the best we can do.

Keep that in mind, and be sure your function doesn't enter infinite loops. Also keep in mind that you may write a function that is correct, but does not have the same precision errors as mine because you have not written the computations in the *exact* same way, and thus will behave slightly differently on the same examples. Such are the joys of working with floating point numbers...

At the end of the day, the following assertion must hold:

```
from math import nextafter as na, inf
def neigh(f): return na(f,inf), na(f,-inf)
assert all( abs(n**.5 - (r:=di(lambda x:x**2-n, 0,99,d)) ) <= d
            or r in neigh(n**.5) # result is closest float
        # for di in [di, di_while]
        for n in range(20) for d in [1e-7, 1e-16, 1e-32] ), r
```

It ensures that either the required precision is achieved, or we are in a case where there is no representable floating-point value between `a` and `b`, like in the case `a=1.414213562373095, b=1.4142135623730951`, and so we lack information to choose between the two.

(24) Now write a function `di_while`, as in the previous questions, but implemented using a **`while`** loop.

For assertions, simply uncomment the line

```
for di in [di, di_while]
```

in the previous question's assertion.

(25) So we have made a big deal of **`float`**'s precision problems in the last question. Is the grass greener with **`int`**?

Write a function `find(x,l)` that returns an index of `x` in the sorted list `l`, if it exists, and `None` if `x` does not appear in `l`.

The fact that `l` is *sorted* must immediately suggest to you to use a dichotomic search! Why write a linear algorithm when you can write a logarithmic one?

If an element appears twice, any suitable index may be returned; it need not be the first or last or anything specific.

The function must implement a dichotomic search and satisfy the following assertions:

```
assert find(3,[3]) == 0

assert all( (fe:=find(ie:=i,ir:=r)) == (i if 0<=i<N else None)
            for N in range(9) for r in [range(N)]
            for i in list(r)+[-1,N] ), (ir, ie, fe)

assert all( l[fe:=find(ie:=i,il:=l)] == i
            for N in range(5) for R in [2,3]
            for l in [[ e for e in range(N) for _ in range(R) ]]
            for i in range(N)), (il, ie, fe)
```

You will code the dichotomic search recursively. Since the bounds a,b are not parameters of the function, you will need to hide them from the user. A good approach taken by Python's library[ae] is to pass them as optional arguments, but this has two drawbacks as a general solution for that type of need: (1) each recursive call will need to pass the same x,l again, which is tolerable here but lacks legibility when there are more arguments to repeat, and (2) sometimes it makes no sense to let the user see and modify the parameters you recurse upon. What you will do instead is use a recursive subfunction, here called z. Your function will thus be of the form:

```
def find(x,l):
    def z(a,b):
        ....
    return z(...)
```

Now, on to the algorithm itself. If you code the search naïvely, you will probably use a,b as inclusive bounds, and you are very likely to run into infinite loop problems because... what is the integral middle of $[\![0, 1]\!]$? Whether you use floor or ceiling, you run into the same "the middle is confused with a bound" problem as in the previous question, except this time, you cannot stop immediately on grounds that maximum possible precision has been achieved.

It is quite possible to write a correct search that way, but it takes a lot of strategically placed +1 and -1 to make it correct, and it is not immediately obvious when reading the code that it is correct. So let's do things a bit differently, so that correctness is more obvious.

We shall take a leaf from the "modern" way of representing ranges, using $[\![a, b[\![$ instead of $[\![a, b]\!]$; that is to say, the lower bound is inclusive, but the upper bound is exclusive. Note that this is the convention adopted by Python's **range** and slice notation, and used in many other languages as well. It has many good properties that make working with ranges easier:

- $b - a$ is the length of the range, not $b - a + 1$,
- 0 and the length of the collection are the starting bounds, not 0 and length -1,
- cutting a range does not require $+1$ or -1 anywhere either: $[\![a, b[\![= [\![a, m[\![+ [\![m, b[\![$.

We will take advantage of that here. We shall write the computation of the middle as

```
m = a + (b-a) // 2
```

that is to say, we keep our starting point, but divide the length b-a of the search space by 2. We test for length 0 and 1 for our stopping conditions, and otherwise, we know

[ae] https://docs.python.org/3/library/bisect.html

that the remaining range, being of length at least 2, will split nicely. Note that it does not matter whether we use `floor` or `ceil` in the computation of $\frac{b-a}{2}$.

Last constraint, do not test equality at every loop. Just one inequality will do, until you only have one element left.

The moral of the story is that, despite having no precision loss problems or even integer overflows (in Python!), `int` is not necessarily simpler to handle than `float`. Some thought is required to handle bounds correctly, and the inclusive/exclusive convention makes it much easier.

Let us test the performance of our function: the code below compares the performance of `find` and that of the default `list.index` method, which performs a linear traversal of the list, as does the `x in l` construct.

```
def time_test():
    from timeit import timeit
    print(f"len\tindex\tdicho\tratio")
    for N in [10**n for n in range(9)]:
        l = list(range(N))
        i = [i*N//10 for i in range(10)]+[N-1]
        ti = sum( timeit(lambda: l.index(k), number=1) for k in i )
        td = sum( timeit(lambda: find(k,l),  number=1) for k in i )
        print(f"{N}\t{ti}\t{td}\t{ti/td}")
```

The generated text isn't pretty but can be pasted into your preferred spreadsheet software.

However you visualise it, you should get something like that:

len	index	dicho	ratio i/d
1	4.92E-06	9.13E-06	0.5388829451
10	6.00E-06	1.71E-05	0.3514938517
100	7.39E-06	1.79E-05	0.4123655766
1000	3.60E-05	2.29E-05	1.57186566
10000	4.36E-04	3.26E-05	13.38771158
100000	0.004439856	4.32E-05	102.6793683
1000000	0.030203839	4.33E-05	698.1933006
10000000	0.272337098	6.73E-05	4046.012322
100000000	2.78551715	8.81E-05	3.16E+04

From this, the following conclusions can be drawn: `index` is slightly more efficient than `dicho` for small lists. Somewhere between lists of size 100 and 1 000, (from more testing, the inflection point seems to be around 625) this changes, and `dicho` becomes orders of magnitude more efficient for large lists. This is exactly what you expect when comparing a simple linear algorithm to a more sophisticated logarithmic one. The latter is more expensive to set up, which does not necessarily pay on small instances, because there is little time there to gain anyway, but crushes the simple algorithm on large instances, where the cost of the increased sophistication is dwarfed by the time gains.

Of course, this discussion elides the question of the cost of making sure the input list is ordered in the first place. . .

If you run `time_test` and your version of `find` does not exhibit the same asymptotic behaviour, you have not correctly implemented the dichotomic search; try again!

35.2 This is all very derivative. . .

functions are nice objects, easy to pass and return
handling optional arguments
difference between symbolic math function and python function object
mathplotlib for interactive visualisation of data

Let's get back to our search for zeroes in continuous functions. A binary search is quite efficient, as we have seen, but we can do better. We could use the slope of the tangent line of the curve at a given point to guide the way, much more efficiently, towards the zero. But before we can do that, we must be capable of computing — and approximation of — that slope.

That is to say, we want to numerically approximate $f'(a)$, the derivative of f at point a. It is defined by the classical formula

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

and can thus be approximated by

$$f'(a) \approx \frac{f(a+h)-f(a)}{h}, \tag{35.1}$$

for small values of h. (35.1) is called the forward difference formula. There are others that can be used to the same effect:

$$f'(a) \approx \frac{f(a)-f(a-h)}{h}$$

is the backward difference formula. And one can average the two and get

$$f'(a) \approx \frac{1}{2}\left(\frac{f(a+h)-f(a)}{h} + \frac{f(a)-f(a-h)}{h}\right) = \frac{f(a+h)-f(a-h)}{2h}, \tag{35.2}$$

which is the central difference formula. We shall use mostly (35.1), because it is the most straightforward, and (35.2), because it is numerically better than the other two, as we shall see.

Let us use as example the functions

```
def f(x): return 2*x
def g(x): return x**2 - 2
```

Notice that $f(x) = g'(x)$.

(26) Write a function `deriv(f, x, h=.01)` that returns the value $f'(x)$, as approximated through (35.1)[p240], using the provided value of h.

The following assertion must hold:

```
assert all( abs(deriv(g,x)-f(x)) <= 0.02 for x in range(100) )
```

(27) Write a function `fderiv(f, h=.01)`, that returns the derivative function f', as approximated by `deriv` (with `h` potentially overridden).

See Sec. 20.5.5[p65]: "Functions are first-class citizens" and Sec. 20.5.6[p66]: "Anonymous functions: **lambda**".

The following assertion must hold:

```
assert all( fderiv(lambda x:x**2)(x) == deriv(lambda x:x**2, x)
            for x in range(100) )
```

(28) Define `G = fderiv(g)`, and let us test how well our approximation performs: we should have $G(x) \approx f(x)$. Write a procedure `test_deriv()` whose invocation yields this:

```
 x   f(x)  G(x)          f(x)-G(x)
-2.0 -4.00 -3.99   -0.01000000000001755
-1.9 -3.80 -3.79   -0.009999999999995346
...
 1.9  3.80  3.81   -0.009999999999999343
 2.0  4.00  4.01   -0.009999999999888765
```

You may want to read Sec. 21.4.9[p88]: "Formatting strings" to format the output properly.

Note: you can't use **float**ing-point numbers in **range**. To understand why, please study Sec. 46[p286]: "Fear the floating-point ranges" (on your own time).

You can see that we have a precision of about 0.01, give or take. You can play with the values of h, going to 0.001, 0.0001, . . . to see how it affects the precision. Does adding more zeroes always increase the precision? Why?

(29) Now go back to the original values of h and modify `deriv` to use the central difference formula (35.2)[p240] instead of (35.1)[p240]. What do you observe? Does adding more zeroes to h still increase the precision, even if just at first?

For your information: There are good reasons for the central difference formula to outperform the others. It can be shown that there are constants K, K', *and* K'', *such that*

$$\left|\frac{f(a+h)-f(a)}{h} - f'(a)\right| \leqslant hK \quad \text{and} \quad \left|\frac{f(a)-f(a-h)}{h} - f'(a)\right| \leqslant hK'\,,$$

while

$$\left|\frac{f(a+h)-f(a-h)}{2h} - f'(a)\right| \leqslant h^2K''\,.$$

Of course, for h *small, that means a much better precision in general for the central difference. . .*

(30) To make this more fun, let's visualise the curves using `matplotlib`. First, you need to install the relevant packages, for your OS and for Python (via `apt`, `pacman`, or `pip3`).

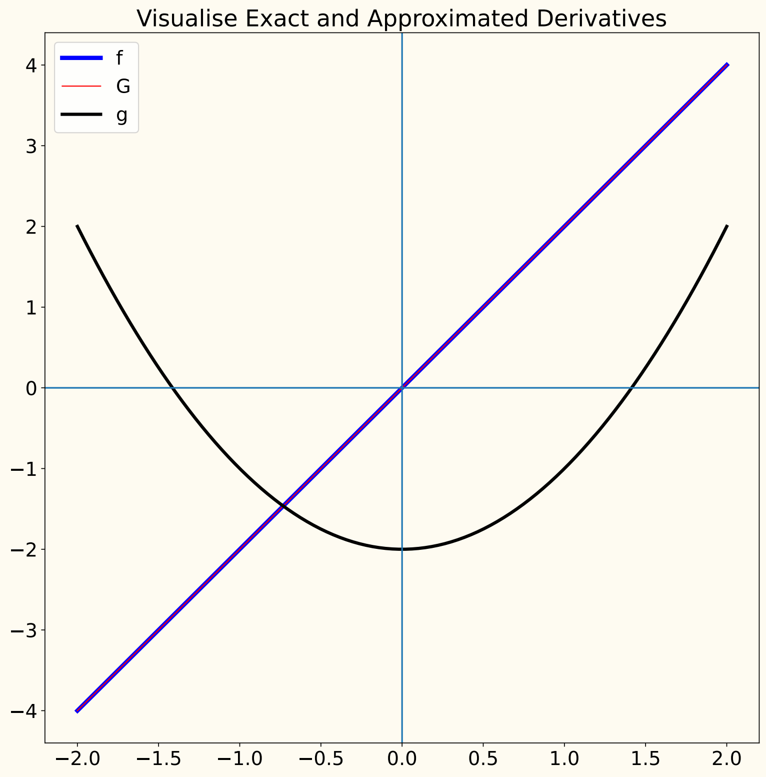


Figure 3: `matplotlib` visualisation

Your best bet is to install the package using your package manager. It should be

```
sudo apt install python3-matplotlib
```

under Debian / Ubuntu and

```
sudo pacman -S python-matplotlib
```

under Arch.

If you're not administrator on your machine or that fails for whatever reasons, run the following command:

```
pip3 install matplotlib
```

If all installs well, good. If not, you may need some libraries for your OS, which again, requires admin access. Under a Kubuntu 21.04 I installed the packages below — you may or may not need to do something equivalent. Pay attention to what `pip3` tells you.

```
sudo apt install libtiff5-dev libjpeg8-dev libopenjp2-7-dev zlib1g-dev \
    libfreetype6-dev liblcms2-dev libwebp-dev tcl8.6-dev tk8.6-dev \
    python3-tk libharfbuzz-dev libfribidi-dev libxcb1-dev
```

When all is installed properly we can start to play. Copy the following code after the function definitions:

```
import numpy as np, matplotlib.pyplot as plt
x = np.linspace(-2,2,100) # x varies in [-2,2], 100 uniform samples
npf = f(x) ; npg = g(x) ; npG = G(x) # our functions, with special object x

plt.figure(figsize=(12,12))
plt.rcParams.update({"font.size": 18 }) # I need glasses, OK?

plt.plot(x,npf,"b"      ,label="f", linewidth=4)
plt.plot(x,npG,"r"      ,label="G", linewidth=1)
plt.plot(x,npg,"black"  ,label="g", linewidth=3)

plt.title("Visualise Exact and Approximated Derivatives")
plt.legend(loc="best")
plt.axvline(0); plt.axhline(0) # draw abscissa and ordinate axes

# plt.savefig("../derivapprox.pdf", transparent=True)
plt.show()
```

When executing that, you should get an interactive version of Figure 3[p242]. The commented `plt.savefig` line is of course what I used to generate the figure — plus a run of `pdfcrop`.

Observe, by zooming on the blue line, that our central approximation, in red inside the thicker blue line, is indistinguishable from the real thing.

You may amuse yourself by trying more complicated functions.

35.3 The Newton–Raphson method

Let us come back to our problem of zeroes of f. How can we use our newfound derivation powers to get even more efficient approximations than with a binary search?

The idea is to start with a guess x_0, then to refine that guess by computing the tangent to f at x_0, then following that tangent to where it intersects with the abscissa, and wherever that is, this is our new and improved guess x_1.

After all, a tangent is simply a linear approximation of f at that point. Instead of finding the zero of f directly, we find that of an approximation.

What happens if $f'(x_0) = 0$? Well, we are stuck, since the tangent is parallel to the abscissa. Unlike the binary search, this method is not *guaranteed* to converge; however, when it does, it does so *very* fast, as we shall see.

Let us take it from the top. We have a differentiable real function f, and an initial guess x_0. The tangent line of f at x_0 is given by the equation

$$t(x) = f'(x_0)(x - x_0) + f(x_0) \,,$$

and crosses the abscissa at x_1, solution of $t(x_1) = 0$:

$$\begin{aligned} 0 &= f'(x_0)(x_1 - x_0) + f(x_0) \\ x_1 &= x_0 - \frac{f(x_0)}{f'(x_0)} \end{aligned}$$

Following the same reasoning, at each step we obtain the next guess by computing

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \,.$$

When do we stop that process? Unlike before, we lack a well-defined search interval, so we cannot know how close we are to the solution. We do know, however, how close $f(x_n)$ gets to zero, and we shall use that as a criterion.

(31) Write a function `newton(f,x,eps=1e-15)` that computes, if possible, a zero of f using the Newton-Raphson method. The function shall trigger an assertion if it runs into a null derivative, and a guess x_n shall be considered good enough to return if $|f(x_n)| < \varepsilon$. Optionally, for debugging purposes, you can have it print each of its guesses.

Tip: this function is doable in three lines — at least in its recursive version.

With it, let us compute $\sqrt{2}$ again, with initial guess 1:

```
>>> res = newton(g,1) # here with optional printing of guesses
-> 1
-> 1.4999999999999996
-> 1.4166666666666667
-> 1.4142156862745099
-> 1.4142135623746899
-> 1.4142135623730951
>>> sqrt(2)
1.4142135623730951
>>> sqrt(2)-res
0.0
```

Note how fast it is, compared to the binary search!

Dichotomy is a general approach; specific problems may allow for more even more efficient specific approaches

36 Comprehension expressions

Read Sec. 23.5[p148]: "Comprehension expressions" with great attention. Do not neglect the examples in Sec. 23.5.3[p153]: "Common comprehension patterns".

For nearly each question, you will use a comprehension expression. For instance, if I ask for the set of all even numbers strictly less than n, then I expect to see

```
{ i for i in range(n) if i%2==0 }
```

If I ask you for a function returning the set of all even numbers strictly less than n, I expect

```
def evens(n):
  return { i for i in range(n) if i%2==0 }
```

Where a comprehension expression is possible, an answer based on usual constructions by iteration will not be suitable for the purpose of this exercise.

comprehension expressions are compact, legible, and easy to write.
you love comprehension expressions

36.1 Warm-up

(32) Write a function cart_prod(A,B) returning the Cartesian product of sets A and B. For instance:

```
>>> cart_prod({'a', 'b'}, {1,2,3})
{('a', 1), ('a', 2), ('a', 3), ('b', 1), ('b', 2), ('b', 3)}
```

Note that, since the result is a set, the order in which its elements are displayed is unpredictable; see Sec. 23.3[p136]: "Sets: class set". If you want to display the elements in a legible order, you can use **sorted(.)**

The following assertions should hold:

```
assert cart_prod(range(3), []) == set()
assert type(cart_prod([1],[2])) is set
assert cart_prod(range(2), range(10, 12)) == {(0,10), (0,11), (1,10), (1,11)}
```

(33) . . . now compute

```
>>> cart_prod("ab", {1,2,3})
```

instead of

```
>>> cart_prod({'a', 'b'}, {1,2,3})
```

What happens, and why? Is it a bad thing?

If in doubt, read Sec. 22.4$_{[p109]}$: "`for .. in .. range` loop" again, especially the part where the keywords iterable *and* collection *appear.*

what works for an iterable type usually should work for another

(34) Write a function `squares(n)` returning the list of all square numbers (i.e. integers that are the square of another integer) in $[\![0, n]\!]$.

As usual, make sure to avoid floating point equality tests; for instance you must not write `sqrt(i) == int(sqrt(i))`. Do not use $i**0.5$ either. No square root or floating point number of any kind.

The following assertion must hold:

```python
assert squares(100) == [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

conditionals in comprehension expressions
comprehension expressions and boolean operators

(35) Write a predicate `isprime(n)` ($\mathbb{N} \to$ `bool`) testing whether a natural integer is prime, that is to say, whether it is strictly greater than 1 and divisible only by 1 and n.

As usual in this exercise, the body must be written in one line of the form `return <expr>`.

Read Sec. 23.5.3.4$_{[p154]}$: "Reductions", especially the part about `any` *and* `all`*.*

The following assertion must hold:

```python
assert [ i for i in range(30) if isprime(i) ] \
     == [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

combine comprehensions with boolean operators

36.2 Palindromes and other one-liners

All of the following functions must be written in one line: that is to say, their body must be of the form `return <expr>`.

You may write a first version of them using normal loops as a draft if it helps you, but the final product must be of this form. Of course the solution will often be a comprehension expression, but sometimes it can be another simple expression.

sequence index manipulation
`any`, `all`, and `sum` reductions

(36) Write a predicate `palindrome(s)` testing whether the sequence `s` is a palindrome.

Read Sec. 21.4.5[p82]: "Slicing and dicing, concatenation, repetition", especially about negative indexes. Read Sec. 23.5.3.4[p154]: "Reductions", especially about `any` and `all`.

A *palindrome* is a sequence that can be read either left-to-right or right-to-left: `ABCBA` is an example.

To test that, you will test that all elements are equal to their mirror: the first to the last, the second to the penultimate, and so on.

The following assertions must hold:

```
assert palindrome('abba')
assert palindrome('abcba')
assert palindrome('')
assert palindrome('a')
assert not palindrome('ab')
```

(37) Write a function `inverse(s)` returning the list of the elements of the sequence `s`, in reverse order. For the purpose of this exercise, you will use a comprehension expression, not a `[::-1]` slice.

The following assertions must hold:

```
assert inverse('abc') == ['c', 'b', 'a']
assert inverse('') == []
```

(38) Write a predicate `palinv(s)` equivalent to `palindrome(s)`, but using `inverse`.

The following assertions must hold:

```
assert palinv('abba')
assert palinv('abcba')
assert palinv('')
assert palinv('a')
assert not palinv('ab')
```

(39) **`for elem in collection` versus `for i in range`**

Write a function `rmfrom(s,bad)` returning the list of the elements of the collection `s` that do not appear in the collection `bad`. The order of elements must be preserved if `s` is a sequence.

The following assertion must hold:

```
assert rmfrom('esope reste ici et se repose', 'aeiouy ') == \
    ['s', 'p', 'r', 's', 't', 'c', 't', 's', 'r', 'p', 's']
```

(40) **laziness is a virtue: reuse previous functions**

Write a function `rmspaces(s)` returning a list of the elements of the sequence `s`, in the original order, from which spaces have been removed.

The following assertion must hold:

```python
assert rmspaces('esope reste ici et se repose') == \
       [ 'e', 's', 'o', 'p', 'e', 'r', 'e', 's', 't',
         'e', 'i', 'c', 'i', 'e', 't', 's', 'e', 'r',
         'e', 'p', 'o', 's', 'e']
```

(41) **are you virtuous yet?**

Write a predicate `palindrome_sentence(s)` testing whether the sentence described by the sequence `s` is palindrome. A sentence is palindrome is the sequence of its letters is palindrome — whitespace is abstracted away.

The following assertions must hold:

```python
assert palindrome_sentence('esope reste ici et se repose')
assert not palindrome_sentence('esope reste ici et se reposes')
```

(42) **a lot of maths can be translated in Python almost directly**

Write a function `fsum(f,i,j)` such that

$$\texttt{fsum(f,i,j)} = \sum_{k=i}^{j} f(k) \; .$$

The following assertions must hold:

```python
assert fsum (lambda i:i, 0,10) == 55
assert fsum (lambda i:i**2, 0,10) == 385
```

37 Our generators have character

(43) *You will need Sec. 21.4.6*[p85]*: "Python strings use Unicode".*

Write a function `crange(a,b)` that returns a generator for all characters from `a` to `b`, in Unicode point order. This can (and must) be done in one line, using a generator expression.

It must satisfy the following assertion:

```
assert "".join(crange('A','Z')) == 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
assert next(crange("a","b")) == "a"
```

generator expressions
variadic functions

Read Sec. 23.5.1[p149]: "Comprehensions for every type; first contact with generators".

(44) *This question requires the* **`yield`** *or* **`yield from`** *keywords from Sec. 28[p202]: "Iterables, iterators, and generators", as well as Sec. 24.1[p165]: "Variadic function definition". It is advised to use Sec. 23.6[p159]: "Packing and unpacking" as well.*

Write a variadic function `charrange`($a_1, b_1, \ldots, a_n, b_n$) returning a generator for all characters of the successive ranges a_k, b_k, as defined in the previous question. It need not be written in one line.

It must satisfy the following assertions:

```
assert "".join(charrange('A','Z','a','z','0','9')) == \
    'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789'
assert next(charrange("a","b")) == "a"
assert "".join(charrange()) == ''
```

`yield` and `yield from`, and the difference between them

38 And then there were Nones. . .

side effects; order of evaluation

Read Sec. 21.5[p93]: "Nihilism: `NoneType`: expression versus statement".

Mentally execute the script below, and write down the output which you expect Python to produce.

```
2+2
print(2+2)
print(print(2+2),print(2+2))
l= [ 1+i for i in range(3) ]
pl = [ print(1+i) for i in range(3) ]
print(l,pl)
```

Execute the code. Compare what was actually produced to what you thought would be. If they do not match exactly, take the time to understand why.

39 Sets, dictionaries and slices training

If you have not already done so, read Sec. 21.4.5[p82]: "Slicing and dicing, concatenation, repetition", Sec. 23.3[p136]: "Sets: class `set`", and Sec. 23.4[p139]: "Dictionaries: class `dict`".

sets are hash tables $\Longrightarrow$ no mutable values
sets are unordered $\Longrightarrow$ not indexable
sequence slicing syntax (on indexable stuff only!)
`False == 0`, so problems in sets, dicts
comprehensions are loops behind the scenes, so side effects work as usual

Mentally execute the following blocs of code, and write down on a piece of paper what you think Python will display.

In cases where Python's output is not entirely predictable, be sure to note that on your answer, and explain the cause and extent of this unpredictability.

Then execute the code and confront your answer to reality.

(45) `print(set('totto'))`

(46) `print({'totto'})`

(47) `print({{'toto'}, {'tata'}})`

(48) `print('abcde'[-1])`

(49) `print({'abcde'}[0][1])`

(50) `print('abcdefg'[2:5])`

(51) `print((list('abcdefg')*3)[2:5])`

(52) `print((list('abcdefg')*3)[19:22])`

(53) `print('abcdefg'[-5:-2])`

(54) `print( list(range(12))[13:5:-2] )`

(55) `print({0:1, None:2, False:5})`

(56)
```
s = { print(i) for i in range(1,3) }
ss = { (i,print(i)) for i in range(1,3) }
sss = { (i,i,print(i)) for i in range(1,3) }
print(s,ss,sss,sep='\n')
```

40 What the *what*!?

Some days, in my profession, you come across code that you just *have* to share with the world. As therapy.

In this exercise, I share with you very *special* code that I have seen written sincerely, candidly, by my own students, in answer to questions in this very document, whether in class or during an exam[af].

The trick is, I don't tell you whether it works, or what question it is supposed to answer. It is up to you to figure out what it does — or purports to do.

Then you must correct it where necessary, and simplify it.

This section is small for now, as I only recently had this idea to systematically weaponise students'… *creativity* into exercises, but, with the upcoming exams, I have every confidence that it will grow fast :-)

(57) Courtesy of a student from 3A STI Apprentissage, 2020-2021, who wishes to remain anonymous:

```python
def spicy_function(X, Y):
 E = set()
 { E.add( (x,y) ) for x in X for y in Y }
 return E
```

This works, and not quite accidentally either, but it's interesting to understand *why*, and to understand what the *value* of

```python
{ E.add( (x,y) ) for x in X for y in Y }
```

is, and what happens to it.

side effects ≠ the value denoted by the comprehension

(58) 3A STI, 2019-2020. This one was written all in one line. Given the limitations of the PDF / paper format, I had to wrap it.

This is sad, as some of the poetry is lost.

```python
True if len(p) == 0 else not False in {True if p[j] ==
          p[len(p)-j-1] else False for j in range (len(p)//2)}
```

(C==True)==True is not better in an expression-if
"there *has* to be a better way to write this"

[af]It's less funny during an exam; no points are awarded for being facepalm-worthy.

41 Encapsulating the sparse matrices

VERY FRESH PAINT (28/09)! Report typos and bugs to me!

encapsulation: hide implementation details behind an interface
tailor your implementation to your needs
test your assumptions regarding performance

A *sparsely populated matrix* (or *sparse matrix* for short) is a matrix in which the vast majority of the values are 0, `null`, `None`, or whatever other value signifies "nothing to see here!" in the context at hand.

For instance, when implementing a game operating on a map of size 100×100, if you have only a couple hundreds of characters on the map, the matrix representing the situation would qualify as sparse. Sparseness, and its opposite, *density*, have strong implications regarding the performance of various implementations of matrices.

In this exercise we shall write two different implementations of matrices, hiding the dirty details of the implementation from the user behind an interface. This extremely common technique is referred to as *encapsulation*[ag] in Object-Oriented Programming (OOP). You will need to read and understand the basics of Sec. 26[p178]: "Object Oriented Programming in Python".

Then, we shall compare the performance of those two interchangeable implementations.

To keep things simple, we are not going to implement matrices in all generality, but only square matrices M_N of size $N \times N$, $N \in \mathbb{N}$, of the form

$$M_N = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & N-1 \end{bmatrix},$$

which is to say, diagonal matrices defined as

$$[M_N]_{ij} = \begin{cases} i & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}.$$

Furthermore, we shall only implement one operation on those matrices: the computation of the sum $\sum M_N$ of their elements:

$$\sum M_N = \sum_{i,j} [M_N]_{ij}.$$

Though we shall not implement that, we shall allow for the possibility of the matrices being modified by the user; thus our representations and the implementation of our methods shall

[ag] https://en.wikipedia.org/wiki/Encapsulation_(computer_programming).

remain fully general, and will not take advantage of the extremely specific form of M_N. It just happens that we initialise our matrices to M_N, that's all, and every bit of code you write should work equally well if we decided to initialise to random values instead.

While this still appears very restrictive, to go back to the "game" application, this is sufficient to *simulate* populating sparse matrices and executing an operation that needs to take everything on the grid into account. With this, we shall have more than enough to draw pretty clear conclusions with regards to performance; you'll see.

Our implementations of matrices shall follow the following interface: the class constructor will take as single argument the dimension N, of default value 100, and the matrix object shall offer:

- ◇ the attribute `N`, storing the dimension,
- ◇ the attribute `m`, storing the internal representation of the matrix, which the user is not really supposed to interact with directly,
- ◇ the method `sum`, taking no argument, returning the sum of the elements of the matrix,
- ◇ and support a string representation (via `repr`, cf. Sec. 26.7[p186]: "String representations `str` and `repr`"), for which I'll provide most of the code.

(59) Let us implement a class `matrix` providing a naïve "list of list" implementation of M_N.

You might want to refresh your reading of Sec. 23.2.2.2[p128]: "Case study: nested lists/ matrices" on the initialisation of matrices.

Complete the following code to initialise the matrix to M_N:

```
class matrix():
    def __init__(s, N=100):
        s.N = N
        s.m = [ [... for ...] for ... ]

    def __repr__(s): return f"matrix({repr(s.m)})"
```

Yes, as the code indicates, populating the internal representation should be done in one (logical) line. Remember that Sec. 22.2[p107]: "Conditional expression: `.. if .. else ..` ternary operator" exists.

You should obtain

```
>>> matrix(3)
matrix([[0, 0, 0], [0, 1, 0], [0, 0, 2]])
```

and the following assertion should hold:

```
assert matrix(3).m == [[0, 0, 0], [0, 1, 0], [0, 0, 2]]
```

Aside on repr: Note that our repr does not exactly follow Python convention, in that it does not return the Python code that would produce the object:

```
>>> matrix([[0, 0, 0], [0, 1, 0], [0, 0, 2]])
TypeError: 'list' object cannot be interpreted as an integer
```

Morally, we should have returned the string "matrix(3)", but we are anticipating a more general version of matrix, where we can initialise a matrix with whatever we want, and modify it.

(60) Now, implement the matrix.sum method, by completing the following code:

```
    def sum(s):
        return sum(...)
```

There again, the implementation should be in one line, following the structure of the code I provide, and the following assertion should hold:

```
assert all( matrix(N).sum() == N*(N-1)//2 for N in range(10) )
```

We're all done with our first implementation.

(61) **Intermission:**

Pop quiz on something completely different: in the assertion above, how do I know that N*(N-1)//2 will be evaluated as (N*(N-1)) // 2, which works because maths guarantee that N*(N-1) is even[(ah)] for all N, and not N * ((N-1)//2) which introduces a rounding error if N-1 is odd?

```
>>> [ (a,b) for N in range(10) if (a:= N*(N-1)//2) != (b:= N*((N-1)//2)) ]
[(1, 0), (6, 4), (15, 12), (28, 24)]
```

Tip: Figure 1[p106].

always think of precedence and associativity
when in doubt, don't write useless parentheses: check, and learn

(62) Now let's go back to our matrices, and provide another implementation, called smatrix, with a completely different internal representation. Spoiler alert, it is called smatrix because it is optimised for sparse matrices.

Instead of a natural "list of lists" representation, we'll use a "mapping" representation, whereby we store a mapping from coordinates of non-zero cells to their value. Any cell not appearing in the mapping is assumed to contain 0.

(ah)The product of an even integer with any other integer is even. Of two consecutive numbers, one is even. Therefore. . .

For instance,

$$M_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

is represented as

$$\{ (1,1) \mapsto 1,\ (2,2) \mapsto 2 \} .$$

You will implement this mapping using a dictionary. More specifically, you will use a `defaultdict`, so that any coordinate not in the dictionary is associated with 0. See Sec. 23.4.1.2[p146]: "`defaultdict`, from `collections`".

You will complete the following code:

```
class smatrix():
    def __init__(s, N=100):
        s.N = N
        s.m = defaultdict(..., { ... for ...  })

    def __repr__(s): return f"smatrix({repr(s.m)})"
```

You should get something like

```
>>> smatrix(3)
smatrix(defaultdict(<function smatrix.__init__.<locals>.
    <lambda> at 0x7f54f0748220>, {(1, 1): 1, (2, 2): 2}))
```

which is not pretty to look at — we'll fix that later — and the following assertions must hold:

```
assert type(smatrix(3).m) is defaultdict
assert smatrix(3).m == { (1, 1): 1, (2, 2): 2 }
assert smatrix(3).m[(999, 999)] == 0
```

(63) Now, implement the `smatrix.sum` method, by completing the following code:

```
    def sum(s):
        return sum(...)
```

As usual, in one line.

For this first implementation, I add the constraint that it must be a pretty naïve implementation of $\sum_{i,j} [M_N]_{ij}$, following the structure of `matrix.sum`, explicitly asking the internal representation for the values of all the cells in the matrix.

If you see a *much* better way of doing this, don't worry, we'll get there in a couple of questions. If you don't, and wonder what I'm talking about, don't worry, your first instinct will probably be the naïve implementation I'm asking for, so all is well :-)

```
assert all( smatrix(N).sum() == N*(N-1)//2 for N in range(10) )
```

(64) Write a method `smatrix.full_matrix()` that returns the "list of lists" representation of the matrix.

You should have

```
>>> smatrix(3).full_matrix()
[[0, 0, 0], [0, 1, 0], [0, 0, 2]]
```

and the following assertion must hold:

```
assert all( smatrix(N).full_matrix() == matrix(N).m for N in range(10) )
```

translate from one representation to another

(65) Now let us fix our `repr` so it doesn't look too ugly. Change what needs to be changed so that we see

```
>>> smatrix(3)
smatrix([[0, 0, 0], [0, 1, 0], [0, 0, 2]])
```

instead of

```
>>> smatrix(3)
smatrix(defaultdict(<function smatrix.__init__.<locals>.
    <lambda> at 0x7f54f0748220>, {(1, 1): 1, (2, 2): 2}))
```

The following assertion should hold:

```
assert all( repr(smatrix(N)) == "s"+repr(matrix(N)) for N in range(10) )
```

hide dirty internal structure from the user

(66) It's time for some performance testing.

Here is some code for performance testing:

```
def test():
    from timeit import timeit
    for desc,f in [
    ("matrix init",             lambda: matrix()),
    ("matrix init+sum",         lambda: matrix().sum()),
    ("sparse matrix init",      lambda: smatrix()),
    ("sparse matrix init+sum", lambda: smatrix().sum()),
        ]:
        print(f"{desc:<25}{timeit(f,number=1000):.3f}")
```

Execute it, and you should see numbers telling a story similar to this:

```
matrix init               0.183
matrix init+sum           0.446
sparse matrix init        0.005
sparse matrix init+sum    1.208
```

At the risk of stating the obvious, these are performance numbers and will vary from machine to machine; they should lead to the same *conclusion*, however.

And what is that conclusion? We see that `smatrix` initialises massively faster than `matrix`. This is not surprising, as the former allocated all cells into memory, even if zero, and almost all of them are zero, whereas the latter only allocates memory for non-zero cells. All good.

However, we are disappointed to see that `sum` performance is actually *worse* for `smatrix`. How come?

good structure + bad algo = bad performance

(67) If you had a better idea for implementing `smatrix.sum`, now is your time to shine. If not, now is your time to think one up.

Tip: just sum over the non-zero values of the internal representation. See Sec. 23.4[p139]: "Dictionaries: class `dict`" to find a useful method of `dict`, beginning by v, that you might want to use. . .

With the new implementation, the performance profile should match

```
matrix init               0.185
matrix init+sum           0.467
sparse matrix init        0.005
sparse matrix init+sum    0.006
```

In other words, `smatrix` is now massively more efficient than `matrix` for all supported operations.

I hope you appreciated in passing how convenient it is that we can just completely change the implementation of a method at the drop of a hat, and still trust in the correctness of the code because we have unit-test assertions in place. Isn't it nice? (You *do* have assertions in place, right? Right?)

good structure + good algo = good performance

(68) What performance profile would you expect from those two implementations on *dense* matrices, that is, matrices in which most values are non-zero?

error 404: universal perfect structure not found

42 Power to the sets!

We recall the notion of **powerset** of a set S, denoted by $\wp(S)$, 2^S, $\mathbb{P}(S)$, $P(S)$, etcetera.

$$s \in \wp(S) \iff s \subseteq S \qquad \text{or} \qquad \wp(S) = \{ s \mid s \subseteq S \} .$$

In other words, the powerset of S is the set of all (non-strict) subsets of S. For instance:

$$\wp(\{0,1\}) \; = \; \left\{ \varnothing, \{0\}, \{1\}, \{0,1\} \right\}$$

Further recall the property $|\wp(S)| = 2^{|S|}$, which is one of the reasons for the use of the notation 2^S: we have $|2^S| = 2^{|S|}$; the other reason is of course the bijection between the powerset and the set of functions $\{0,1\}^S$ — both reasons boil down to the same thing, in the end.

It is interesting to see a proof of that property:

Theorem 3 (Cardinality of Powerset)**.** *Let S be a set; then $|\wp(S)| = 2^{|S|}$.*

Proof sketch. By induction on $|S|$.

If $|S| = 0$, then $S = \varnothing$, and $|\wp(\varnothing)| = |\{\varnothing\}| = 1 = 2^0$.

Let $S = \{e\} \cup T$, with $|T| = n$, and assume $|\wp(T)| = 2^n$. Let $s \subseteq S$; then either $e \in s$ or $e \notin s$, and in either case $s - \{e\} \in \wp(T)$. So we have in total 2^n powersets containing e, and 2^n not containing e, so $|S| = 2^{n+1} = 2^{|S|}$. □

Note that the proof suggests a recursive definition of the powerset. . .

(69) Define a function `powerlist(l)` returning the list of the positional sublists of the list (or any other iterable) `l`, in no particular order.

By "positional sublist", I mean a list containing some elements of `l`, based on their positions, not on their values. Thus, if values are repeated in the list `l`, they are still treated as distinct.

The following assertions must hold:

```python
assert powerlist([]) == [[]]
assert sorted(powerlist([1,2,3]), key=lambda x:(len(x), x)) == \
    [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
assert sorted(powerlist([1,1,1]), key=len) == \
    [[], [1], [1], [1], [1, 1], [1, 1], [1, 1], [1, 1, 1]]
assert all( len(powerlist(range(n))) == 2**n for n in range(5) )
```

You will implement this recursively and take care to avoid repeating redundant recursive calls.

This can easily be done in three lines (excluding function declaration).

The cardinality proof above hints at the recursion you need to apply.

You may — no, you most definitely *will* — want to apply some pattern-matching or packing/unpacking, here: see Sec. 22.6[p113]: "Pattern matching: **match..case**" and Sec. 23.6[p159]: "Packing and unpacking". Hint: what does `e,*l = l` or `*l,e = l` do?

using packing/unpacking for recursion on lists

(70) Now write `powerlist2`, another version of `powerlist`, implemented non-recursively.

There again, this can easily be done in three lines (excluding function declaration).

It must satisfy the same assertions as `powerlist`; you can have the assertions apply to both functions like so:

```
for powerlist in (powerlist, powerlist2):
    assert powerlist([]) == [[]]
    assert sorted(powerlist([1,2,3]), key=lambda x:(len(x), x)) == \
        [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
    assert sorted(powerlist([1,1,1]), key=len) == \
        [[], [1], [1], [1], [1, 1], [1, 1], [1, 1], [1, 1, 1]]
    assert all( len(powerlist(range(n))) == 2**n for n in range(5) )
```

a recursive definition can also suggest an easy loop implementation

(71) Can you implement a function `powerset(s)` in Python that naively matches $\wp(\cdot)$ and returns a `set` of `set`s? Why?

Tip: Sec. 23.3[p136]: "Sets: class `set`", Sec. 23.3.1[p138]: "Frozen sets: class **`frozenset`**".

(72) Implement a function `powerset(s)` corresponding to $\wp(\cdot)$, and returning a `set` of **`frozenset`**s. The parameter s may be of any iterable type, and must not be altered by the call.

Reminder: packing/unpacking works on sets, and `set.pop` exists as well.

The following assertion must hold:

```
assert all( powerset(r:=range(n)) == { frozenset(s) for s in powerlist(r) }
            for n in range(5) )
```

sets of sets are common in maths; need thought in Python

(73) Write a generator function `powergen(s)`, where s is again any iterable, than generates all sub`set`s of s.

The following assertions must hold:

```
assert type(powergen([])) is type(_ for _ in [])
assert all( type(s) is set for s in powergen(range(5)) )
assert all( set(map(frozenset, powergen(r:=range(n))))
            == { frozenset(s) for s in powerlist(r) }
            for n in range(5) )
```

You can use **`yield from`** if you want, but in that instance it is simpler not to.

recursion in generator functions

43 Let's get primitive!

In Sec. 35.2[p240]: "This is all very derivative. . . ", we computed numerical approximations of derivatives. Let us now do the same thing for integral calculus and primitives.

apply comprehension expressions to
solve a seemingly nontrivial problem in a few lines

Our goal is to compute a numerical approximation of a primitive (or antiderivative) of any given continuous function of type $\mathbb{R} \to \mathbb{R}$. That is to say, given a function f, we want to obtain a function F such that $F' \approx f$.

Because constants disappear during derivation, every function f has infinitely many primitives, differing only up to an additive constant. Each can be written F or $\int f(x)\,dx$.

Primitives and definite integrals, which represent the area beneath the curve of the function, are related by the following equation:

$$\int_a^b f(x)\ dx \quad = \quad F(b) - F(a)\ .$$

Thus, if we can compute a definite integral, we can obtain a suitable primitive via

$$F(x) \quad = \quad \int f(x)\,dx \quad = \quad \int_0^x f(t)\ dt\ .$$

Our first objective will therefore be to implement definite integrals. The simplest way to do that is to use Riemann integrals.

The general idea is to approximate the area under the curve by partitioning the abscissa into many smaller intervals, and approximate the area under the curve on each smaller interval by a rectangle. Then you can sum all rectangles. The smaller each small interval, the better the approximation.

There are different possible choices regarding the selection of the smaller intervals, and of the value used for the height of each rectangle, but they all give the same result when the intervals get infinitesimally small. This is illustrated visually in Figure 4[p261].

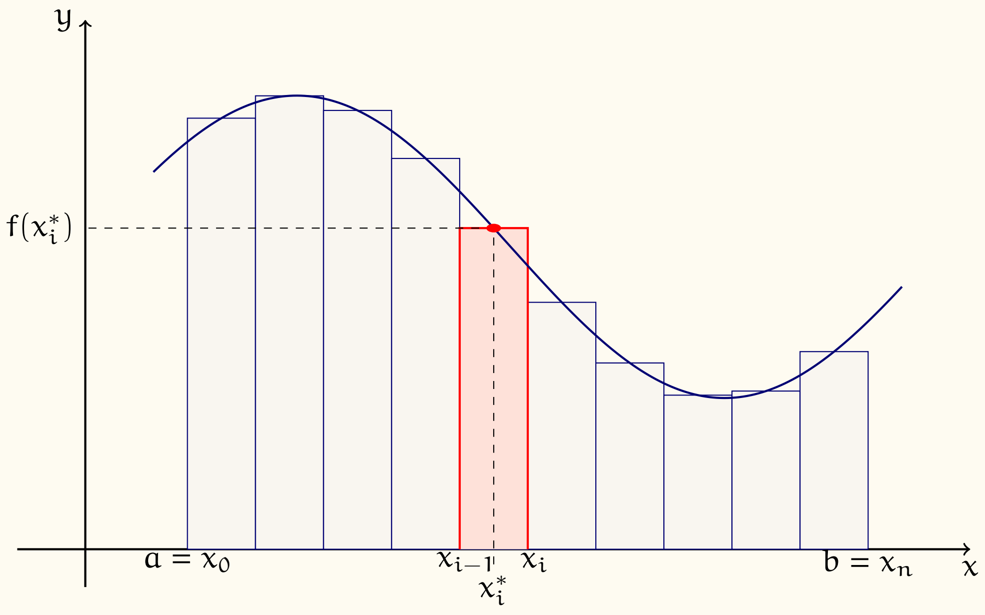


Figure 4: General idea of Riemann integration

Definition 4 (Riemann integral)**.** Let $f : [a, b] \to \mathbb{R}$ and

$$P = \left[(x_0, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n)\right] ,$$

a partition of $[a, b]$. That is to say, we have

$$a = x_0 < x_1 < x_2 < \cdots < x_n = b$$

The **Riemann sum** of f on P is

$$\sum_{i=1}^{n} f(x_i^*)\, \Delta x_i ,$$

where $\Delta x_i = x_i - x_{i-1}$ and $x_i^* \in [x_{i-1}, x_i]$.

The **Riemann integral** is the limit of the Riemann sum as the maximal size of partitions $\|\Delta x\| = \max_i \Delta x_i$ goes to 0:

$$\int_a^b f(x)\, dx \quad = \quad \lim_{\|\Delta x\| \to 0} \sum_{i=1}^{n} f(x_i^*)\, \Delta x_i .$$

The choice of x_i^* can produce different sums, but does not matter at the limit, for the integral.

For our purposes, we shall take uniform partitions, that is to say all Δx_i are equal, and select x_i^* as the middle of its interval:

$$x_i^* \quad = \quad \frac{x_i + x_{i-1}}{2}, \qquad \forall i ,$$

this is called a **middle Riemann sum**.

(74) Write a function `partition(a,b,n)` that, given two floating point values a and b, and $n \in \mathbb{N}$, returns the partition of the interval $[a, b]$ into n equal parts.

Specifically, you must return a list of couples `[ (a1, b1), ..., (an, bn) ]` such that $a_1 = a$, $b_n = b$, and all intervals $[a_i, b_i]$ are of the same size.

It should be written in one or two lines using a comprehension expression, and it must satisfy the following assertions:

```python
assert partition(-1, 1, 1) == [(-1.0, 1.0)]
assert partition(0,30,3)   == [(0.0, 10.0), (10.0, 20.0), (20.0, 30.0)]
assert partition(30,0,3)   == [(30.0, 20.0), (20.0, 10.0), (10.0, 0.0)]
assert partition(-1, 1, 4) == [(-1.0, -0.5), (-0.5, 0.0),
                               (0.0, 0.5), (0.5, 1.0)]
```

(75) **Applying `for a,b in l`: multiple assignment in iteration**

Write a function `riemann(f, a, b, n=100)` that returns an approximation of

$$\int_a^b f(x)\,dx\,,$$

the definite integral of f on the interval $[a, b]$, performed by middle Riemann sum, with a partition of $[a, b]$ in n equal parts.

This function can and should be written in one line, of the form `return sum ( ... )`, and should satisfy the following assertions:

```python
assert abs(riemann(lambda x:x, 0,1) - 0.5) < 1e-9
assert abs(riemann(sqrt, 0,1) - 2/3) < 1e-4
```

(76) Write a function `primitive(f, x, n=100)` returning the value

$$F(x) \quad = \quad \int_0^x f(t)\,dt\,,$$

as approximated by a middle Riemann sum on a uniform partition in n equal parts.

The following assertion must hold:

```python
assert all( abs( primitive(lambda x:x, x) - (x*x / 2)) < 1e-9
                 for x in [-32, 0, 1, 2, 8, 64] )
```

(77) Write a function `fprimitive(f, n=100)` returning F, the approximated primitive function of f, as per the previous question.

The following assertion must hold:

```python
assert all( fprimitive(sqrt)(x) == primitive(sqrt, x) for x in range(100) )
```

(78) Let us visualise this. Using the same example functions f and g as in Sec. 35.2[p240]: "This is all very derivative. . . ", graph the functions g, $g+2$, and $F = \int f(x)\,dx$, as given by `F = fprimitive(f)`.

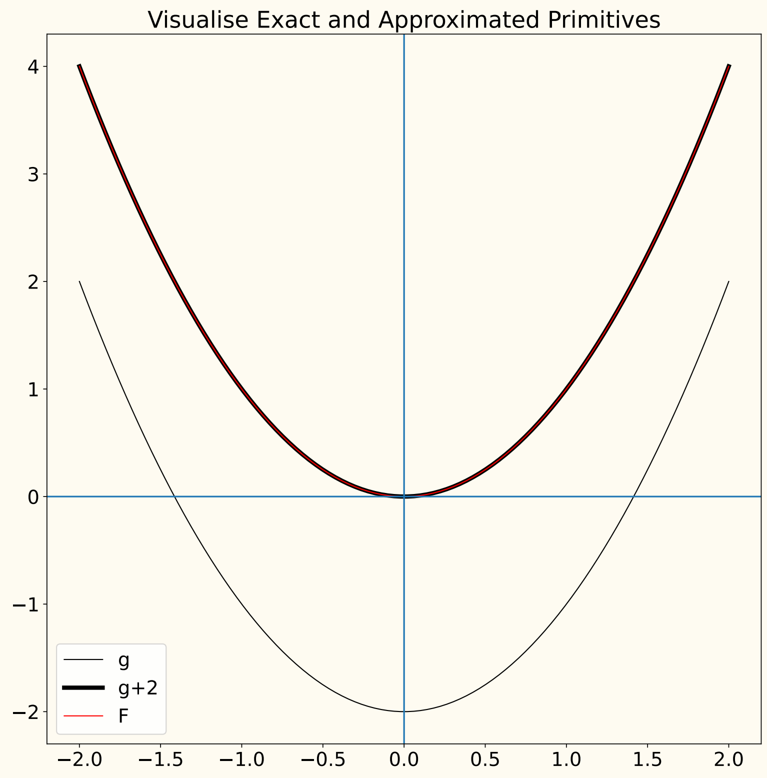


Figure 5: `matplotlib` visualisation of primitive

You should obtain something similar to the nearby figure. You should, as for derivatives, see that the error due to the numerical approximation is imperceptible.

We shall come back to derivation in Sec. 44[p263]: "A smidgen of Computer Algebra", using symbolic methods instead of numerical ones. Note that computing *primitives* symbolically is a *vastly* more difficult problem than computing *derivatives* symbolically. Many primitives of rather simple functions do not even have a closed form! For instance,

$$\int e^{-x^2}\,dx, \qquad \int x^x\,dx, \qquad \int \frac{1}{\ln x}\,dx, \qquad \int \sin x^2\,dx, \qquad \int \frac{\sin x}{x}\,dx, \ldots$$

Thus, we shall *not* come back to integration from a symbolic standpoint.

44 A smidgen of Computer Algebra

*This exercise requires a good understanding of Sec. 22.6[p113]: "Pattern matching: **`match..case`**", Sec. 26.5[p183]: "**`match`**ing attributes", and Sec. 27[p196]: "Advanced structural pattern **`match`**ing". For the second section, Sec. 26[p178]: "Object Oriented Programming in Python" is also required.*

apply advanced pattern matching to a rather complex problem

In Sec. 35.2[p240]: "This is all very derivative. . . ", we used numerical methods to approximate the values of derivatives at any point, which is all well and good but. . . didn't you wish we could get exact answers? For instance

$$\frac{d}{dx}\left(x^2 - 2\right) = 2x\,,$$

or

$$\frac{d}{dx}\left(1 + 2\ln(x^2 - 1)\right) = \frac{4x}{x^2 - 1}$$

or

$$\frac{d}{dx}\left(\ln x(3x^2 + 1)\right) = 3x + \frac{1}{x} + 6x\ln x\,.$$

How do we get *that* kind of answers out of the computer? Well, one way is to fork out the money for Computer Algebra Systems (CAS) such as Maple, Mathematica, or MATLAB, or install free and open-source alternatives such as SageMath[ai], Axiom, or Maxima, or even just look up the solution on `https://www.wolframalpha.com/` . . . but where is the fun in *not* reinventing the wheel?

Instead, we shall implement our own rudimentary CAS. We will need to manipulate mathematical expressions symbolically to compute derivative functions.

First step, what is the formal grammar of those expressions? We shall limit ourselves to

$$e \;::=\; x \;\mid\; e+e \;\mid\; e-e \;\mid\; e \div e \;\mid\; e \times e \;\mid\; f(e) \;\mid\; e^e \;\mid\; -e \;\mid\; \nu\,,$$

where $\nu \in \mathbb{R}$ is a constant value, x is a variable name and f a function name. In practice we shall support only ln.

44.1 A perfect `match`

(79) **implementing an inductive type definition in Python**

Define a class system in the style of Sec. 27[p196]: "Advanced structural pattern `match`ing" for the type of mathematical expressions.

Note that every construct is binary, even function calls and exponentiation that are not typically thought of as "operators". Thus I suggest defining a larger type `BinExpr` to handle all of theses, and only defining $+, -, \times, \div$ as proper `BinOp`. This will enable the factorisation of some rules. Thus I propose that you copy this

[ai]If we had a *serious* Python project involving symbolic computation, the sane thing would be to use Sage and SymPy. SymPy is a Python library for symbolic computation, and Sage, which includes SymPy, is partly implemented in Python, and interoperates with it.

```python
@dataclass
class BinExpr:
    a: object
    b: object

class BinOp (BinExpr): pass

@dataclass
class UnOp:
    a: object
```

and define each operator by inheritance of those types. Variable and function names shall be strings.

For instance, you should be able to write

```python
>>> x = "x" # our main variable name
>>> f1 = Plus(1, Mul(2,Call("ln",Minus(Pow(x,2),1))))
>>> f1
Plus(a=1, b=Mul(a=2, b=Call(a='ln', b=Minus(a=Pow(a='x', b=2), b=1))))

>>> f2 = Mul(Call("ln",x), Plus(Mul(3,Pow(x,2)), 1))
>>> f2
Mul(a=Call(a='ln', b='x'), b=Plus(a=Mul(a=3, b=Pow(a='x', b=2)), b=1))
```

Make sure to define `f1`, `f2`, and `x` in your source file, we shall use them as running examples.

(80) **implementing recursive functions on inductive types**

Ok, we have formulæ, but they are ugly to look at. Write a function `estr` such that

TODO ADD ASSERTIONS THROUGHOUT EXERCISE

```python
>>> estr(f1)
'(1 + (2 * (ln(x^2 - 1))))'

>>> estr(f2)
'((ln x) * ((3 * x^2) + 1))'
```

Recall the trick from Sec. 27[p196]: "Advanced structural pattern **match**ing" to use an attribute `symb` to associate a symbol to each operator.

There are still a lot of parentheses, but analysing operator precedence to get rid of some of them would be a more difficult exercise. This is good enough for our purposes.

(81) **embedding semantics in inductive type definitions**
catching specific exceptions

We have symbolic expressions, which is nice. But at some point we want numerical results as well, if only to be able to graph them. Write a function[aj] `eval(e,var,val)` that produces the numerical value of the evaluation of the expression `e` when the variable `var` is affected the value `val`.

For instance, you should obtain:

```
>>> eval(f2, x, 0)
inf

>>> eval(f2, x, 0.1)
-2.371662645783867

>>> eval(f2, x, 1)
0.0

>>> eval(f2, x, 2)
9.010913347279288

>>> eval(f2, x, 1.4)
2.3149289879539445

>>> eval(f2, "y", 1)
... an error of some sort
```

For graphing purposes, in case of division by zero or domain error — for instance ln is not defined everywhere — you will return the value `float('inf')`, which is to say ∞. For this, you will need to use `try/except`. Note that you should be precise in which exceptions you catch:

```
>>> log(0)
ValueError: math domain error

>>> 1/0
ZeroDivisionError: division by zero
```

We specifically want to intercept `math domain error`, not all `ValueError`s. `ValueError`s can arise in many other cases, in fact it is probably a `ValueError` which we want to raise if we evaluate an improper expression. To avoid catching unwanted exceptions, use something like

```
except ValueError as e:
    if str(e) =="math domain error":
        return float('inf')
    raise e
```

[aj]Note that a function named `eval` is already defined in Python, but it does something quite different — it evaluates a string containing Python code. There is no harm in masking its definition, as we do not use it.

Thus, we do not interfere with our ability to raise `ValueError` when faced with an expression which we cannot evaluate:

```
>>> eval([],x,1)
ValueError: []
```

Note that it is possible to use the same trick as for string conversion to handle all `BinExpr` in one line. Just as we have an attribute `symb` that contains the symbol of a construct, we can have an attribute `sem` that contains its semantics.

For instance I have

```
class Plus       (BinOp):
    symb = "+"
    sem = lambda x,y:x+y
```

You can do the same thing not only for all operators, but also for `Call` and `Pow`.

There is a niggling little difficulty to it, though. In Sec. 27[p196]: "Advanced structural pattern **match**ing", we could write directly

```
match e:
    case BinOp(l,r):    return f"({fstr(l)} {e.symb} {fstr(r)})"
```

Since `sem` is a function, and not a constant like `symb`, when `BinOp` (or `BinExpr`) is instantiated it becomes a bound method, and thus takes `self` as a first argument. Recall that `e.sem(a,b)` is a notational shortcut for `BinExpr.sem(e,a,b)`, if `e` is of type `BinExpr`.

To avoid this, you need to get the attribute `sem` not from the *instance* `e`, but from the *type* `BinExpr`. Thus you will write something like `type(e).sem(a,b)`.

If you run into the error

```
TypeError: Call.<lambda>() takes 2 positional arguments but 3 were given
```

That is probably the origin of the problem.

Another potential difficulty is the handling of `Call`. We restrict ourselves to things like `Call("ln", e)`, where the left-hand side is a constant function name — and we have only need of ln, specifically, but would like to be able to extend to sin, cos, etc. You need to write a semantics attribute of the form $\lambda f, e : [\![f]\!](e)$, where, for instance, $[\![\cdot]\!]$:`"ln"`$\mapsto$ ln. Think carefully about how to do that.

Or, you could just write one line per operator and not have to think about any of that, but that's just no fun at all.

Recall as well the trick of using `z = lambda e: eval(e,var,val)` for the recursive calls, explained at the end of Sec. 27.1[p201]: "An overlong aside on naming conventions".

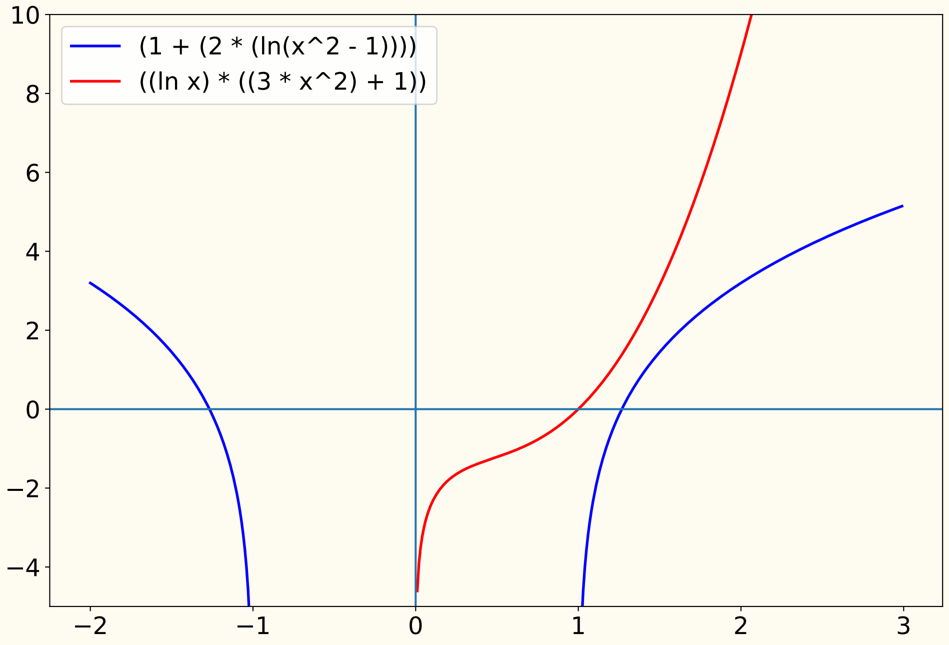
(82) Now that **eval** is all set, let us graph our functions. This time we do not get to cheat with numpy to define the functions — recall the magical x in question (30)[p242]:

```
x = np.linspace(-2,2,100) # x varies in [-2,2], 100 uniform samples
npf = f(x)
```

— we do it the hard way instead, by generating sequences of couples (x,f(x)), and graphing that.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(12,8))
plt.rcParams.update({"font.size": 18 })

X = [-2 + i/100 for i in range(500) ]
Yf1 = [eval(f1,x,X) for X in X]
Yf2 = [eval(f2,x,X) for X in X]
plt.ylim([-5, 10]) # limit the y axis
plt.plot(X,Yf1,"b",label=estr(f1), linewidth=2)
plt.plot(X,Yf2,"r",label=estr(f2), linewidth=2)

plt.legend(loc="best")
plt.axvline(0); plt.axhline(0)

##plt.savefig("../excasf1f2.pdf", transparent=True)
plt.show()
```

You should obtain this:



(83) Now we can move on to the very heart of the matter: symbolically computing the derivative. Your goal is to write a function D(e,x), where e is an expression and x a variable name — in practice "x" — that returns an expression for $\frac{d}{dx}e$, the derivative of e with respect to x.

For instance, we should get

```
print("f2:", estr(f2), "\n\t->")
print(estr(D(f2,x)))
----------------------------------------------------------------------
f2: ((ln x) * ((3 * x^2) + 1))
  ->
(((1 / x) * ((3 * x^2) + 1)) + ((ln x) * (((0 * x^2)
                                              + (3 * (2 * x^1))) + 0)))
```

For now we shall not make any attempt at simplifying the expressions thereby obtained — e.g. multiplications and additions by 0 — that will be the goal of the next question.

To achieve the computation of the derivation, recall (some of) the rules of derivation. We have:

$$\begin{aligned} \frac{d}{dx}c &= 0 \qquad c \in \mathbb{R} \\ \frac{d}{dx}x^n &= nx^{n-1} \\ \frac{d}{dx}\ln x &= \frac{1}{x} \end{aligned}$$

as well as, using f' as short for $\frac{d}{dx}f(x)$:

$$\begin{aligned} (cf)' &= cf' \qquad c \in \mathbb{R} \\ (f+g)' &= f' + g' \\ (f-g)' &= f' - g' \\ (fg)' &= f'g + fg' \end{aligned}$$

Those rules should be enough to handle f2. For f1, you will also need to support the chain rule, or composition rule:

$$(g \circ f)'(x) = g'(f(x))f'(x)\ .$$

Don't do that for now, we shall come back to it later, once we have a complete chain for f1...

(84) We can now compute derivatives, and they are correct, but there are many obvious simplifications left on the table. We want to define a function simp(e) to simplify the expressions e we obtain.

Simplifying mathematics equations is actually a very difficult topic in all generality, where most questions become undecidable. Indeed there are so many ways to manipulate the expressions, and no clear criterion of when the expression is "fully simplified ". Think of all the possibilities when applying associativity and commutativity rules to all operators, and so on. That way lies madness.

We shall instead only pick up on the most obvious simplifications involving the constants 0 and 1. For instance, we want to obtain:

```
print(estr(D(f1,x)))
print(estr(Df1 := simp(D(f1,x))))
----------------------------------------------------------------------
(((1 / x) * ((3 * x^2) + 1)) + ((ln x) * (((0 * x^2)
                                                    + (3 * (2 * x^1))) + 0)))
(((1 / x) * ((3 * x^2) + 1)) + ((ln x) * (6 * x)))
```

Even this is not fully straightforward to code. Consider the expression

$$(0 \times x^2) + (3 \times e) ,$$

where e is some sub-expression. When doing your recursive descent into the structure, you only see something of the form

```
Plus( Mul(..), Mul(..) )
```

you don't know yet that the left-hand side is zero, so you cannot simplify. You will need to come back later, from the top, and do another pass. The alternative would be to write deep patterns, like

```
Plus( Mul(0, e)), Mul(..) )
Plus( Mul(e, 0)), Mul(..) )
...
```

but the number of rules explodes exponentially with the number of simplifications you want to detect as well as the depth of detected patterns. This is not sustainable.

So, you are not going to write `simp` immediately. First, write a function `fixpoint(f,e)` that applies a function `f` on `e` repeatedly, until a fixed point e^* is reached: that is, until $f^n(e) = f^{n+1}(e) = e^*$. It then returns e^*.

In other words, `f` is applied on `e` until it can find nothing left to change.

(85) Now that we have `fixpoint`, we can code `simp`. The idea is that we shall have the architecture suggested at the end of Sec. 27.1[p201]: "An overlong aside on naming conventions":

```
def simp(e):
    def z(e):
        match e:
            case Plus(0,e) | Plus(e,0): return e
            ...
    return fixpoint(z,e)
```

The sub-function z does only one pass, but it is applied repeatedly until all simplifications are exhausted. For the patterns themselves, start with the simplest identity function you can write, then add the special patterns — `Plus(0,e)` etc — on top. Make it so.

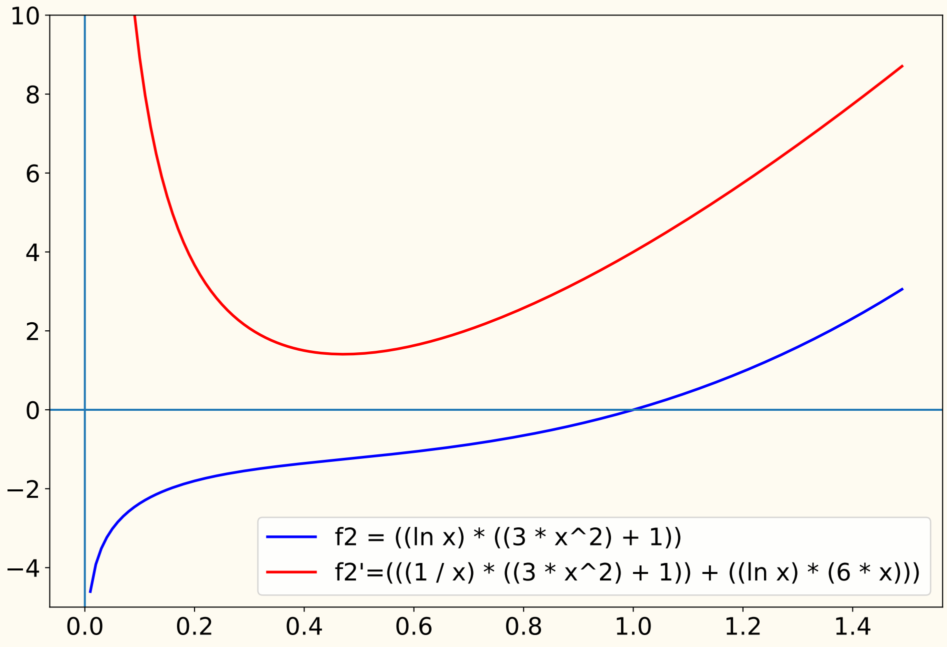
We obtain

```
(((1 / x) * ((3 * x^2) + 1)) + ((ln x) * (6 * x)))
```

for the derivative of `f2`, which matches

$$\frac{d}{dx}\left(\ln x(3x^2+1)\right) \quad = \quad 3x + \frac{1}{x} + 6x\ln x\,.$$

(86) Using our **eval**, `simp`, and `D` functions, produce a plot of `f2` and its derivative:



(87) Now let us deal with `f1`. Recall that $g \circ f(x) = g(f(x))$ and that we need the chain rule

$$(g \circ f)'(x) = g'(f(x))f'(x)\,,$$

which we are going to write more compactly as

$$(g \circ f)' = (g' \circ f)f'\,.$$

How does that apply here? We have $\ln(x^2-1)$; $g = \ln$ and $f = \lambda x : x^2 - 1$. Let us simplify that view and consider any expression as a function — by default, a function of x.

Under that view, we have two expressions $g = \ln x$ and $f = x^2 - 1$. We already have the machinery necessary to derive either expression. There remains to implement the composition $\circ$.

It is actually very simple: $g \circ f$ it is the substitution of all instances of x in g by the expression of f. In our example:

$$g \circ f \;=\; (\ln x)[x \leftarrow f] \;=\; (\ln x)\left[x \leftarrow x^2 - 1\right] \;=\; \ln(x^2 - 1)\,.$$

Likewise we can compute

$$\begin{aligned}(g' \circ f)f' &= \left((\ln x)'\right)\left[x \leftarrow x^2 - 1\right] \cdot (x^2 - 1)' \\ &= \left(\frac{1}{x}\right)\left[x \leftarrow x^2 - 1\right] \cdot 2x \\ &= \frac{1}{x^2 - 1} \cdot 2x \\ &= \frac{2x}{x^2 - 1}\,,\end{aligned}$$

and indeed

$$\frac{d}{dx}\ln(x^2 - 1) \;=\; \frac{2x}{x^2 - 1}\,.$$

That means we have already all the tools we need except for a substitution function. Let us remedy that.

Write a function `sub(e,x,f)` that returns the expression obtained by substituting in `e` every instance of `x` by the expression `f`. For instance, we should have

```
>>> estr( sub(Minus(Mul(2,x),x), x, Plus(1, Pow(x,3))) )
'((2 * (1 + x^3)) - (1 + x^3))'
```

This is not a difficult function to write: it is the identity function, with just one more rule.

(88) Now you can extend the differentiation function `D` to support the chain rule. All you need is a single new **`case`** line.

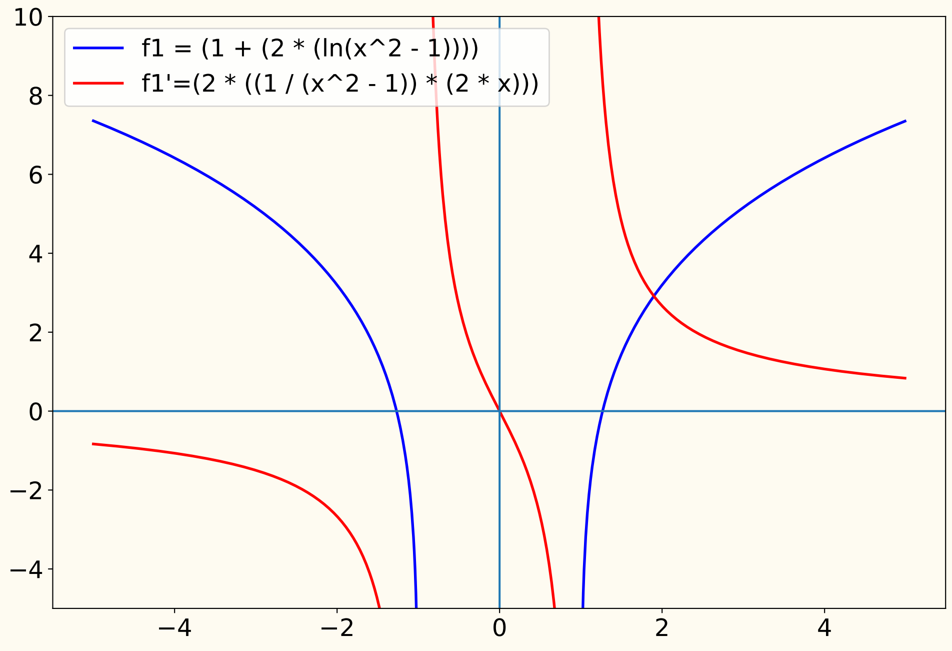
You should obtain

```
print("f1:", estr(f1), "\n\t->")
print(estr(D(f1,x)))
print(estr(Df1 := simp(D(f1,x))))
----------------------------------------------------------
f1: (1 + (2 * (ln(x^2 - 1))))
  ->
(0 + ((0 * (ln(x^2 - 1))) + (2 * ((1 / (x^2 - 1))
                                     * ((2 * x^1) - 0)))))
(2 * ((1 / (x^2 - 1)) * (2 * x)))
```

which is as expected:

$$\frac{d}{dx}\big(1 + 2\ln(x^2 - 1)\big) \quad = \quad \frac{4x}{x^2 - 1} \ .$$

(89) Now that all is said and done, plot `f1` and its derivative:



44.2 I `object`!

Reading Sec. 26[p178]: "Object Oriented Programming in Python" is required for this part of the exercise.

OO wrappers around procedural/functional implementations

Let us make our CAS more user-friendly by setting up a layer of object-oriented syntactic sugar around it. The goal is to set up a wrapper class `F` — for **F**ormula — around our expression type, so that the user can employ the usual syntax to define symbolic expressions. For instance, we should be able to write

```
X  = F('x')                           # declare a symbolic variable
ln = lambda x: F(Call("ln",x.f))      # declare a symbolic function

FF1 = 1 + 2*ln(X**2 - 1)

-----------------------------------------------------------------

>>> FF1
(1 + (2 * (ln(x^2 - 1))))

>>> F1(X,2)
3.1972245773362196
```

```
>>> FF1.D(X)
(2 * ((1 / (x^2 - 1)) * (2 * x)))
```

(90) Begin by creating a class `F` that acts as a wrapper for string conversion. We should be able to do

```
>>> f1
Plus(a=1, b=Mul(a=2, b=Call(a='ln', b=Minus(a=Pow(a='x', b=2), b=1))))

>>> F(f1)
(1 + (2 * (ln(x^2 - 1))))

>>> F(f1).f  # the expression is stored internally as attribute f
Plus(a=1, b=Mul(a=2, b=Call(a='ln', b=Minus(a=Pow(a='x', b=2), b=1))))

>>> repr(F1)
'(1 + (2 * (ln(x^2 - 1))))'

>>> str(F1)
'(1 + (2 * (ln(x^2 - 1))))'
```

(91) Extend the class to support

```
>>> F1 + F1
((1 + (2 * (ln(x^2 - 1)))) + (1 + (2 * (ln(x^2 - 1)))))
```

(92) Extend the class to support

```
>>> F1 + 10
((1 + (2 * (ln(x^2 - 1)))) + 10)
```

(93) Extend the class to support

```
>>> 10 + F1
(10 + (1 + (2 * (ln(x^2 - 1)))))
```

(94) At this point imagine what the code is going to look like once you support every operator. There is some factorisation to do. Write a "dispatch" function `disp(op,s,o)` and a function `rdisp(op,s,o)` so that your implementation of + support looks like

```
class F:
    ...
    def  __add__(s,o): return  disp(Plus,s,o)
    def __radd__(s,o): return rdisp(Plus,s,o)
```

(95) Using this, quickly add support for `*, **, -`, so that we can handle

```
X  = F('x')                          # declare a symbolic variable
```

```
ln = lambda x: F(Call("ln",x.f))     # declare a symbolic function

FF1 = 1 + 2*ln(X**2 - 1)

-----------------------------------------------------------------

>>> FF1
(1 + (2 * (ln(x^2 - 1))))
```

(96) Extend the class so that we can write

```
>>> FF1.D(X)
(2 * ((1 / (x^2 - 1)) * (2 * x)))
```

(97) Extend the class so that we can write

```
>>> F1(X,2)
3.1972245773362196
```

instead of

```
>>> eval(f1,x,2)
3.1972245773362196
```

You should be getting the idea by now. . . Using these techniques, we can completely hide our underlying datatype from the end user.

(98) **(Perspectives)** The exercise stops there, but there is no end to the interesting things we could do to improve and extend our CAS. Extensive automatic simplification, handling of integration, an interactive mode where the user chooses which rules to apply to their system, LaTeX output and display, and so on, and so forth.

If you are interested in this, that can be the object of an "Application Projet" at the end of the year — one week full-time projects done in groups of four. Ask me about it if that kind of thing is your cup of tea.

45 Conway sequence: generating fun

Completing this exercise requires a good understanding of Sec. 28$_{[p202]}$*: "Iterables, iterators, and generators", in particular Sec. 28.4*$_{[p207]}$*: "Understanding deeply lazy computations".*

lazy evaluation = performance (often)
implementing lazy evaluation in a complex problem

In this section, we shall play with *Conway* sequences[(ak)], also called *look-and-say* sequences. Mostly, we shall focus on the Conway sequence with seed $C_0 = 1$. Here are the first few elements of this sequence:

$$\begin{aligned}
C_0 &= 1 \\
C_1 &= 11 \\
C_2 &= 21 \\
C_3 &= 1211 \\
C_4 &= 111221 \\
C_5 &= 312211 \\
C_6 &= 13112221 \\
C_7 &= 1113213211 \\
C_8 &= 31131211131221 \\
C_9 &= 13211311123113112211 \\
C_{10} &= 11131221133112132113212221 \\
C_{11} &= 3113112221232112111312211312113211 \\
&\ldots
\end{aligned}$$

How is it defined? C_{n+1} is defined recursively from C_n as the sequence of numbers obtained by reading the digits of C_n out loud, organised by groups of identical digits, announcing first the number of digits, then the digit in each group.

For instance:

- ◇ 1 is read as "one 1" : 11.
- ◇ 11 is read as "two 1s" : 21.
- ◇ 21 is read as "one 2, followed by one 1" : 1211.
- ◇ 1211 is read as "one 1, one 2, and two 1s" : 111221.
- ◇ 111221 is read as "three 1s, two 2s, and one 1" : 312211.
- ◇ and so on...

We want not only to generate this sequence, but to do so efficiently, getting only the first few digits of each number up to a high rank, even though the length of C_n grows exponentially with respect to n. It is clear that the last digits of, say, C_{10} are not involved in the computation of the *first* digits of C_{11}, so if that's what we really need, why compute C_{10} all the way?

Of course, we shall also write a more traditional, sequential implementation as well, for comparison purposes.

(ak)Following Stigler's law of eponymy, Conway sequences are actually due to... errrr, ok, Conway really *did* invent that. The exception to the rule, I guess. Never mind, then. Carry on.

You may not use anything from itertools, as I ask to to reimplement some of its functionality.

(99) **mixing yield and return in a generator function using enumerate**

Write a function upto(g,i), returning a generator for the first i elements generated by g.

Note: *prior to version 3.7, this could and should have been done in one line. This is no longer the case due to changes in the semantics of generator expressions.*

The following assertions must hold:

```
assert next(upto(range(3),8)) == 0

assert list(upto((x for x in range(3)),8)) == [0, 1, 2]

assert list(upto((n*n for n in range(100)),7)) == \
    [0, 1, 4, 9, 16, 25, 36]
```

(100) For fun, write a function nth(g,n), returning the n-th element of an iterator g.

This can and should be done in one line.

The following assertion must hold:

```
assert all( nth(g,i) == i*i
            for i in range(7)
            for g in [(n*n for n in range(7))] )
```

(101) Write a function powers(f,s), where f is a unary function, that returns a generator for the successive powers $s,\ f(s),\ f^2(s),\ f^3(s), \ldots$, where

$$\begin{aligned} f^0(x) &= \lambda x.x && \text{(identity function)} \\ f^n(x) &= f \circ f^{n-1},\ n > 0 \end{aligned}$$

The following assertion must hold:

```
assert next(powers(lambda x:2*x,1)) == 1

assert list(upto(powers(lambda x:2*x,1),7))  == \
    [1, 2, 4, 8, 16, 32, 64]
```

(102) Write a function group(l), with l being an iterable, that returns a generator for the groups of successive identical elements appearing in l. Each group shall be returned as a list.

The following assertions must hold:

```
assert next(group('a')) == ['a']

assert list(group('')) == []

assert list(group('a')) == [['a']]

assert list(group('aaba')) == [['a', 'a'], ['b'], ['a']]

assert list(group('aabbbcdaaaa')) == \
    [['a', 'a'], ['b', 'b', 'b'], ['c'], ['d'], ['a', 'a', 'a', 'a']]
```

This is a simpler version of `itertools.groupby`.

(103) For our purposes, it is probably more efficient to generate the groups as couples `(length,element)` rather than as lists of identical elements. Write a function `groupn(l)`, similar to `group(l)`, but generating said couples.

The following assertions must hold:

```
assert next(groupn('a')) == (1, 'a')

assert list(groupn('aabbbcdaaaa')) == \
    [(2, 'a'), (3, 'b'), (1, 'c'), (1, 'd'), (4, 'a')]
```

(104) For fun, bridge the gap between `groupn` and `group` by writing a function `groupl` that takes as input the output of `groupn`, and converts it into the output of `group`.

This can and should be done in one line.

The following assertions must hold:

```
assert next(groupl(groupn('aa'))) == ['a', 'a']

assert all ( tuple(group(s)) == tuple(groupl(groupn(s)))
             for s in ('','a','aaba','aabbbcdaaaa') )
```

(105) Using `groupn` — since it is the most efficient — write a function `say(s)` that transforms any string `s` into its "look-and-say" version. That is to say, a function that transforms a string representing C_n into a string representing C_{n+1}.

This can and should be done in one line.

The following assertion must hold:

```
assert list(upto(powers(say,'1'),7)) == \
    ['1', '11', '21', '1211', '111221', '312211', '13112221']

assert list(upto(powers(say,'22'),7)) == \
    ['22', '22', '22', '22', '22', '22', '22']
```

(106) Write a procedure `conway(seed='1',maxrnk=100,maxlen=30)` that displays the Conway sequence of seed `seed`, up to and including rank `maxrnk`.

As elements of the sequence grow exponentially in size, we truncate their display to the first `maxlen` digits.

The output of a call of `conway()` should look like this:

```
  0   1
  1   11
  2   21
  3   1211
  4   111221
  5   312211
  6   13112221
  7   1113213211
  8   31131211131221
  9   13211311123113112211
 10   11131221133112132113212221
 11   3113112221232112111312211312 11...
...
 60   132113213221133112132123123112...
 61   111312211312111322212321121113...
```

Spoiler alert: the display should begin to slow down around rank `50`, and slow down to a crawl around rank `60`, making it impractical to go much farther.

To understand why, recall that our implementation of say needs the whole of C_n to begin computing C_{n+1}. C_{50} has 1 166 642 digits; C_{60} has 16 530 884. Keeping up with this quickly becomes impractical.

Yet, we only need a few digits from the beginning of each element, and it is clear, from the way the sequence is constructed, that those depend only on the first few digits of the previous ranks. Thus we only actually make use of an infinitesimal fraction of the digits we compute.

To exploit that fact, we shall overhaul our computation to make sure that there are generators every step of the way.

(107) Write a function `sayg(s)` playing the same role as `say`, except that instead of taking and returning strings, it takes an iterable and returns a generator.

This can and should be done in one line.

The following assertion must hold:

```
assert list(sayg(''))       == []
assert list(sayg('1'))      == ['1', '1']
assert list(sayg('1211'))   == ['1', '1', '1', '2', '2', '1']
```

(108) Write a function `nthpowerg(f,n,s)`, where f is a unary function, n an integer, and s a seed value, that returns

- ◇ a single-value generator for s if $n = 0$
- ◇ an iterator for $f^n(s)$ otherwise, assuming f is an iterator function.

This can be done either iteratively or recursively.

The following assertion must hold:

```
assert "".join( upto(nthpowerg(sayg,6,'1'),8) ) == '13112221'
```

(109) Write a procedure `conwayg`, equivalent to `conway`, but using generators exclusively to achieve the same result. This time, performance should not be an issue. A call to `conwayg()` should look like this:

```
  0   1
  1   11
  2   21
  3   1211
  4   111221
  5   312211
  6   13112221
  7   1113213211
  8   31131211131221
  9   13211311123113112211
 10   11131221133112132113212221
 11   3113112221232112111312211312 11...
 12   1321132132111213122112311311 22...
...
 60   1321132132211331121321231231 12...
 61   1113122113121113222123211211 13...
...
 99   1321132132211331121321231231 12...
100   1113122113121113222123211211 13...
```

and take no time at all.

Note that we have achieved this considerable speedup without in any way lessening the generality of our code, or increasing its complexity — excepting the fact the generators require somewhat more abstraction from the programmer.

Conventional programming could possibly achieve a similar speedup by simply computing the first digits in a fixed-length (`maxlen`) array. This should work, because the sequence visibly "inflates" with each step, so that the `maxlen` first digits of each rank can be computed with *at most* `maxlen` digits of the previous one.

However, we would need to prove that mathematically to have confidence in such code. Moreover, we would need to do so for all possible seeds. Furthermore, we shall see later on that this would be futile because you *cannot* prove that: experimentally, we quickly find values of `maxlen` for which the hypothetical property stated above is simply false. Perhaps we could show that `maxlen` plus some constant would work? Perhaps it holds for all values of `maxlen` greater than some constant M? I don't know.

And still, even if it worked, which it doesn't, it would be inefficient when asking for a large amount of digits from high ranked values, as we would compute many unneeded digits.

And of course, while such an approach *might* plausibly have worked – or could maybe be tweaked into working – for Conway sequences, it would flat out fail with any sequence deprived of anything resembling this supposed inflation property, whereas, using generators, we simply *don't have to care* about the behaviour of the sequence. We know our code is correct, in the sense that we know we are going to compute what we need, and hopefully no more.

There is a cost, of course: a little more thinking is required to manipulate generators correctly, and there is an overhead computational cost to having all those objects messaging each other saying "hey! wake up! I need a value!". If you need *all* the values *all* the time anyway, there is no point in using this; but if not, it is usually a very good investment to make your code as generator-friendly as possible.

(110) Let us quantify the gains from using generators. More specifically, supposing we want to get the first j digits of C_R, the questions are:

◇ How many digits need I compute, manually, to obtain that, globally and for each previous rank $k \leqslant R$?

◇ How many digits are actually computed by the generator-based method?

◇ How many digits are computed by the non-generator method?

◇ What is the ratio between those quantities?

First, as an example, let us see manually what is strictly needed to get the first digit of C_5:

$$\begin{aligned}
C_0 &= \underline{1} \\
C_1 &= \underline{11} \\
C_2 &= \underline{21} \\
C_3 &= \underline{121}1 \\
C_4 &= \underline{1112}21
\end{aligned}$$

$$C_5 \;=\; \underline{3}12211$$

To compute C_5's 3, we need four digits from C_4, as we cannot conclude as to the number of ones until we see a *different* digit.[al]

To compute C_4's 1112, we need 121 from C_3; indeed, without the final one from C_3, we don't know whether C_4 begins with 1112 or 1122 or 1132, etc. And so on, you get the idea.

Back to the generators. Write a procedure `perf(R,j)` evaluating the performance of generator-based versus classical approach on the computation of the first j digits of C_R. The output must look like this:

```
>>> perf(5,1)
Performance analysis: rank 5, 1 digit.
First 1 digit of C_5 = '3'
C_0 : 0 of 0
C_1 : 2 of 2
C_2 : 2 of 2
C_3 : 3 of 4
C_4 : 4 of 6
C_5 : 1 of 6
Total: 12 of 20, or 60.0%
```

For instance the line `C_4 : 4 of 6` means that generators computed four digits of C_4, whereas the classical approach computed all 6 — the classical approach computes the entirety of each rank. In total, generators computed 12 digits, whereas the classical approach computed 20. The seed is ignored in both counts.

Note that the generator approach should exactly match the manual reasoning above!

On higher ranks, you should get:

```
>>> perf(55,30)
Performance analysis: rank 55, 30 digits.
First 30 digits of C_55 = '111312211312111322212321121113'
C_0 : 0 of 0
..
C_4 : 6 of 6
..
C_10 : 5 of 26
..
C_20 : 4 of 408
..
C_30 : 4 of 5808
```

[al]Actually, we could show mathematically that no repetition longer than three can occur, and conclude upon seeing the third one. However, we haven't shown that, our `say` and `sayg` algorithms do not take that into account, and thus for now we don't know whether 1111 may occur, so we need to see the next digit.

```
..
C_40 : 5 of 82350
..
C_50 : 12 of 1166642
C_51 : 12 of 1520986
C_52 : 17 of 1982710
C_53 : 20 of 2584304
C_54 : 21 of 3369156
C_55 : 30 of 4391702
Total: 343 of 18858434, or 0.0018188148602370697%
```

Tips:

Implementing this is a bit tricky.

I advise creating lists to store the needed numbers of digits, one for generators and one for the classical approach, and writing "hacked" versions of `nthpowergp` and `saygp` so that the list concerned with generators is updated each time a digit is computed, as a side-effect.

Note that these hacked versions of `nthpowergp` and `saygp` can and should be subfunctions of `perf`.

(111) Find a simple counterexample for our earlier hopeful assertion that perhaps

> "The j first digits of each rank can be computed with *at most* j digits of each of the previous ranks."

(112) *Perspectives:* for those of you interested in going (much) farther in your understanding of lazy evaluation, I recommend implementing the Conway sequence in Haskell. Haskell is a pure functional language with lazy evaluation.

An implementation of Conway every bit as powerful as our generator version can be obtained completely transparently in just a few lines of code.

Of course, this is *far* outside the scope of this class; or of your curriculum, for that matter. You will not be taught Haskell — or OCaml, or Scheme (Lisp), or indeed any functional language — at INSA CVL. I would recommend studying this in your own time if (and only if) you wish to acquire a larger understanding of programming paradigms and techniques, and are not afraid of maths and abstraction.

Part IV

Additional Python Exercises

The following are retired exercises, made redundant by new material, as well as archives from various tests and non-INSA training sessions. They are provided in no particular order.

They can provide additional fun to any student who may prematurely run out of stuff to do during classes... Idle hands are the Devil's playthings, after all.

Part V[p328]: "DIU EIL: Récursivité" contains even *more* additional exercises, specifically geared around the concept of recursiveness, which you are encouraged to practice on.