3

Programming and Data Structures

The two preceding chapters have introduced mathematical computations using one-line commands, but Sage also allows writing programs with sequences of instructions.

The Sage computer algebra system is in fact an extension of the Python[1] computer language, and allows, except a few differences, to use the Python programming constructs.

The commands described in the previous chapters show that it is not necessary to know the Python language to use Sage; this chapter explains how to use the Python programming structures within Sage. Since we only present basic programming, this chapter can be skipped by the reader fluent in Python; the examples are chosen among the most classical ones encountered in mathematics, so that the reader will quickly grasp the Python programming constructs, by analogy with known programming languages.

This chapter presents in particular the paradigm of *structured programming* with loops and tests, then describes functions dealing with lists and other data structures.

3.1 Syntax

3.1.1 General Syntax

The instructions are generally processed line per line. Python considers the sharp symbol "`#`" as the beginning of a comment, until the end of the line. The semicolon "`;`" splits several instructions on the same line:

[1] Sage 7.6 uses Python 2.7, which only slightly differs from Python 3.

Python language keywords	
`while, for...in, if...elif...else`	loops and tests
`continue, break`	early exit of a code block
`try...except...finally, raise`	deal and raise with exceptions
`assert`	debugging condition
`pass`	no-effect statement
`def, lambda`	definition of a function
`return, yield`	return of a value
`global, del`	scope and deleting variables and functions
`and, not, or`	boolean operations
`print`	text output
`class, with`	object-oriented and context programming
`from...import...as`	library access
`exec...in`	dynamic code evaluation

TABLE 3.1 – General syntax of the Sagecode.

```
sage: 2*3; 3*4; 4*5          # one comment, 3 results
6
12
20
```

In the terminal, a command can be written on several lines by putting a backslash "\" before each end of line, the end-of-line characters being considered as blank characters:

```
sage: 123 + \
....: 345
468
```

An identifier — i.e., a variable or function name, etc. — is formed from letters, digits and the underline symbol "_", and cannot start by a digit. The user identifiers should differ from the language keywords, given in Table 3.1, and which form the core of the Python language. The list of keywords is available by:

```
sage: import keyword; keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'exec', 'finally', 'for', 'from',
'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass',
'print', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

In addition to those keywords, we have the constants `None` (empty value, named `NULL` in other languages), `True` and `False`, and several predefined functions by Python or Sage like `len`, `cos` and `integrate`. It is better not to use those as variable names, otherwise some functionalities might no longer be available. The interpreter knows some additional commands, like `quit` to exit the Sage session. We will discover other commands like `time` or `timeit` later in this book.

Some symbols have a special meaning in Sage. They are explained in Table 3.2.

Sage special symbols and their main uses	
, ;	argument and instruction separators
:	beginning of an instruction block
.	decimal point, access to an object fields
=	assignment of a value to a variable
+ - * /	basic arithmetic operations
^ **	power
% //	quotient and remainder of Euclidean division
+= -= *= /= **=	arithmetic operations with modification of a variable
== != <> is	equality tests
< <= > >=	comparisons
& | ^^ << >>	set operations and bitwise logical operations
#	comment (until end of line)
[...]	construction of a list, access to an element by its index
(...)	function or method call, non-mutable tuples
{...:...}	dictionary construction
\	special character escape (and linear algebra)
@	applying a decorator to a function
?	help access
_ __ ___	last three results

TABLE 3.2 – General Sage syntax (following).

3.1.2 Function Calls

To evaluate a function, its arguments should be put inside parentheses like in `cos(pi)` or in the function call without argument `reset()`. However the parentheses are superfluous for a command: the instructions `print(6*7)` and `print 6*7` are equivalent[2]. The name of a function without argument nor parenthesis represents the function itself and performs no computation.

3.1.3 More About Variables

As seen previously, Sage denotes the assignment of a value to a variable by the equal sign "`=`". The expression to the right of the equal sign is first evaluated, then its value is saved in the variable whose name is on the left. Thus we have:

```
sage: y = 3; y = 3 * y + 1; y = 3 * y + 1; y
31
```

The three first assignments change the value of the variable `y` without any output, the last command prints the final value of `y`.

The `del x` command discards the value assigned to the variable `x`, and the function call `reset()` recovers the initial Sage state.

Several variables can be assigned *simultaneously*, which differs from *sequential* assignments `a = b; b = a`:

```
sage: a, b = 10, 20 # (a, b) = (10, 20) and [10, 20] are also possible
sage: a, b = b, a
```

[2]In Python 3, `print` is a function and thus requires parentheses.

```
sage: a, b
(20, 10)
```

The assignment `a, b = b, a` is equivalent to swapping the values of `a` and `b` using an auxiliary variable:

```
sage: temp = a; a = b; b = temp    # equivalent to: a, b = b, a
```

The following trick swaps the values of `a` and `b` without any auxiliary variable, using additions and subtractions:

```
sage: x, y = var('x, y'); a = x ; b = y
sage: a, b
(x, y)
sage: a = a + b ; b = a - b ; a = a - b
sage: a, b
(y, x)
```

The instruction `a = b = c = 0` assigns the same value, here 0, to several variables; the instructions `x += 5` and `n *= 2` are respectively equivalent to `x = x+5` and `n = n*2`.

The comparison between two objects is performed by the double equal sign "`==`":

```
sage: 2 + 2 == 2^2, 3 * 3 == 3^3
(True, False)
```

3.2 Algorithmics

The paradigm of *structured programming* consists in designing a computer program as a finite sequence of instructions, which are executed in order. Those instructions can be atomic or composed:

- an example of atomic instruction is the assignment of a value to a variable (cf. §1.2.4), or a result output;
- a composed instruction, like a loop or a conditional, is made up from several instructions, themselves atomic or composed.

3.2.1 Loops

Enumeration Loops. An enumeration loop performs the same computation for all integer values of an index $k \in \{a, \ldots, b\}$: the following example[3] outputs the beginning of the multiplication table by 7:

```
sage: for k in [1..5]:
....:     print 7*k  # block containing a single instruction
```

[3] When using Sage in a terminal, such a block of instructions must be ended by an additional empty line, which will be implicit in the whole book. This is not necessary when using Sage through a web browser.

```
7
14
21
28
35
```

The colon symbol ":" at the end of the first line starts the instruction block, which is evaluated for each successive value 1, 2, 3, 4 and 5 of the variable `k`. At each iteration, Sage outputs the product $7k$ via the `print` command.

In this example, the repeated instruction block contains a single instruction (namely `print`), which is indented to the right with respect to the `for` keyword. A block with several instructions has its instructions written one below the other, with the same indentation.

The block positioning is important: the two programs below, which differ in the indentation of a single line, yield different results.

```
sage:  S = 0                        sage:  S = 0
sage:  for k in [1..3]:             sage:  for k in [1..3]:
...      S = S+k                    ...      S = S+k
sage:  S = 2*S                      ...      S = 2*S
sage:  S                            sage:  S
```

On the left the instruction `S = 2*S` is executed only once at the end of the loop, while on the right it is executed at every iteration, which explains the different results:

$$S = (0+1+2+3)\cdot 2 = 12 \qquad S = ((((0+1)\cdot 2)+2)\cdot 2+3)\cdot 2 = 22.$$

This kind of loop will be useful to compute a given term of a recurrence, cf. the examples at the end of this section.

The syntax `for k in [a..b]` for an enumeration loop is the simplest one and can be used without any problem for 10^4 or 10^5 iterations; its drawback is to explicitly construct the list of all possible values of the loop variable before executing the iteration block, however it manipulates Sage integers of type `Integer` (see §5.3.1). Several `..range` functions allow iterations with two possible choices. The first choice is: either construct the list of possible values before starting the loop, or determine those values together with the loop iterations. The second choice is between Sage integers[4] (`Integer`) and Python integers (`int`), those two integer types having slightly different properties. In case of doubt, the `[a..b]` form should be preferred.

While Loops. The other kind of loops are the *while* loops. Like the enumeration loops, they execute a certain number of times the same sequence of instructions; however, here the number of repetitions is not known *a priori*, but depends on a condition.

[4]The commands `srange`, `sxrange` and `[...]` also work on rational and floating-point numbers: try `[pi,pi+5..20]` for example.

Iterations functions of the `..range` form for `a`, `b`, `c` integers	
`for k in [a..b]: ...`	constructs the list of Sage integers $a \leq k \leq b$
`for k in srange (a, b): ...`	constructs the list of Sage integers $a \leq k < b$
`for k in range (a, b): ...`	constructs a list of Python integers (`int`)
`for k in xrange (a, b): ...`	enumerates Python integers (`int`) without explicitly constructing the corresponding list
`for k in sxrange (a, b): ...`	enumerates Sage integers without constructing a list
`[a,a+c..b], [a..b, step=c]`	Sage integers $a, a+c, a+2c, \ldots$ as long as $a+kc \leq b$
`..range (b)`	equivalent to `..range (0, b)`
`..range (a, b, c)`	sets the iteration increment to c instead of 1

TABLE 3.3 – The different enumeration loops.

The *while* loop, as its name says, executes instructions while a given condition is fulfilled. The following example computes the sum of the squares of non-negative integers whose exponential is less or equal to 10^6, i.e., $1^2 + 2^2 + \cdots + 13^2$:

```
sage: S = 0 ; k = 0          #        The sum S starts to 0
sage: while e^k <= 10^6:     #            e^13 <= 10^6 < e^14
....:     S = S + k^2        #           accumulates the squares k^2
....:     k = k + 1
sage: S
819
```

The last instruction returns the value of the variable `S` and outputs the result:

$$S = \sum_{\substack{k \in \mathbb{N} \\ e^k \leq 10^6}} k^2 = \sum_{k=0}^{13} k^2 = 819, \qquad e^{13} \approx 442413 \leq 10^6 < e^{14} \approx 1202604.$$

The above instruction block contains two assignments: the first one accumulates the new term, and the second one moves to the next index. Those two instructions are indented in the same way inside the `while` loop structure.

The following example is another typical example of *while* loop. For a given number $x \geq 1$, it searches the unique integer $n \in \mathbb{N}$ satisfying $2^{n-1} \leq x < 2^n$, i.e., the smallest integer with $x < 2^n$. The program below compares x to 2^n, whose value is successively 1, 2, 4, 8, etc.; it performs this computation for $x = 10^4$:

```
sage: x = 10^4; u = 1; n = 0          # invariant: u = 2^n
sage: while u <= x: n = n+1; u = 2*u  # or n += 1; u *= 2
sage: n
14
```

As long as the condition $2^n \leq x$ is satisfied, this program computes the new values $n+1$ and $2^{n+1} = 2 \cdot 2^n$ of the two variables `n` and `u`, and stores them in place of n and 2^n. The loop ends when the condition is no longer fulfilled, i.e., when $x < 2^n$:

$$x = 10^4, \qquad \min\{n \in \mathbb{N} \mid x < 2^n\} = 14, \qquad 2^{13} = 8192, \qquad 2^{14} = 16384.$$

Note that the body of a *while* loop is never executed when the condition is false at the first test.

As seen above, small command blocs can be typed on a single line after the colon ":", without creating a new indented block starting at the next line.

Aborting a loop execution

The `for` and `while` loops repeat a given number of times the same instructions. The `break` command inside a loop interrupts it before its end, and the `continue` command goes directly to the next iteration. Those commands thus allow — among other things — to check the terminating condition at every place in the loop.

The four examples below determine the smallest positive integer x satisfying $\log(x+1) \leq x/10$. The first program (top left) uses a `for` loop with at most 100 tries which terminates once the first solution is found; the second program (top right) searches the smallest solution and might not terminate if the condition is never fulfilled; the third (bottom left) is equivalent to the first one with a more complex loop condition; finally the fourth (bottom right) has a useless complex structure, whose unique goal is to exhibit the `continue` command. In all cases the final value `x` is 37.0.

```
for x in [1.0..100.0]:
  if log(x+1)<=x/10:  break
```

```
x=1.0
while log(x+1)>x/10:
  x=x+1
```

```
x=1.0
while log(x+1)>x/10 and x<100:
  x=x+1
```

```
x=1.0
while True:
  if log(x+1)>x/10:
    x=x+1
    continue
  break
```

The `return` command (which ends the execution of a function and defines its result, cf. §3.2.3) offers yet another way to early abort from an instruction block.

Application to sequences and series. The `for` loop enables us to easily compute a given term of a recurrent sequence. Consider for example the sequence (u_n) defined by

$$u_0 = 1, \qquad \forall n \in \mathbb{N}, \quad u_{n+1} = \frac{1}{1+u_n^2}.$$

The following program yields a numerical approximation of u_n for $n = 20$; the variable `U` is updated at each loop iteration to change from u_{n-1} to u_n according to the recurrence relation. The first iteration computes u_1 from u_0 for $n = 1$, the second one likewise from u_1 to u_2 when $n = 2$, and the last of the n iterations updates U from u_{n-1} to u_n:

```
sage: U = 1.0                # or U = 1. or U = 1.000
sage: for n in [1..20]:
....:   U = 1 / (1 + U^2)
sage: U
0.682360434761105
```

The same program with the integer `U = 1` instead of the floating-point number `U = 1.0` on the first line will perform exact computations on rational numbers; then u_{10} becomes a rational number with several hundreds digits, and u_{20} has hundreds of thousands digits. Exact computations are useful when rounding errors accumulate in numerical approximations. Otherwise, *by hand* or *with the computer*, the computations on numerical approximations of a dozen digits are faster than those on integers or rational numbers of thousand digits or more.

The sums or products admitting recurrence formulas are computed the same way:

$$S_n = \sum_{k=1}^{n} (2k)(2k+1) = 2\cdot 3 + 4\cdot 5 + \cdots + (2n)(2n+1),$$
$$S_0 = 0, \qquad S_n = S_{n-1} + (2n)(2n+1) \quad \text{for } n \in \mathbb{N} - \{0\}.$$

The following programming method follows that of recurrent sequences; starting from 0, we add successive terms for $k = 1$, $k = 2$, ..., until $k = n$:

```
sage: S = 0 ; n = 10
sage: for k in [1..n]:
....:     S = S + (2*k) * (2*k+1)
sage: S
1650
```

This example highlights a general method to compute a sum; however, in this simple case, a symbolic computation yields the general answer:

```
sage: n, k = var('n, k') ; res = sum(2*k*(2*k+1), k, 1, n)
sage: res, factor(res)    # result expanded, factorized
(4/3*n^3 + 3*n^2 + 5/3*n, 1/3*(4*n + 5)*(n + 1)*n)
```

Those results might also be obtained with the *pen and pencil* method from well-known sums:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}, \qquad \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6},$$
$$\sum_{k=1}^{n} 2k\,(2k+1) = 4\sum_{k=1}^{n} k^2 + 2\sum_{k=1}^{n} k = \frac{2\,n\,(n+1)(2n+1)}{3} + n(n+1)$$
$$= \frac{n(n+1)\big((4n+2)+3\big)}{3} = \frac{n(n+1)(4n+5)}{3}.$$

Example: approximation of sequence limits. While the enumeration loop is well suited to compute a given term of a sequence or series, the *while* loop is adapted to approach numerically the limit of a sequence.

If a sequence $(a_n)_{n\in\mathbb{N}}$ converges to $\ell \in \mathbb{R}$, the terms a_n are *close* to ℓ for n *large enough*. It is thus possible to approximate ℓ by a given term a_n, and the mathematical problem reduces to find a bound for the error $|\ell - a_n|$. This bound is trivial for two sequences $(u_n)_{n\in\mathbb{N}}$ and $(v_n)_{n\in\mathbb{N}}$ such that

$$\begin{cases} (u_n)_{n\in\mathbb{N}} \text{ is increasing,} \\ (v_n)_{n\in\mathbb{N}} \text{ is decreasing,} \\ \lim\limits_{n\to+\infty} v_n - u_n = 0. \end{cases} \tag{3.1}$$

In this case,

$$\begin{cases} \text{the two sequences converge to the same limit } \ell, \\ \forall p \in \mathbb{N} \quad u_p \le \lim\limits_{n\to+\infty} u_n = \ell = \lim\limits_{n\to+\infty} v_n \le v_p, \\ \qquad \left|\ell - \frac{u_p+v_p}{2}\right| \le \frac{v_p-u_p}{2}. \end{cases}$$

A mathematical analysis shows that the two following sequences satisfy the above properties and converge to $\sqrt{ab}$ when $0 < a < b$:

$$u_0 = a, \qquad v_0 = b > a, \qquad u_{n+1} = \frac{2\,u_n v_n}{u_n + v_n}, \qquad v_{n+1} = \frac{u_n + v_n}{2}.$$

The common limit of those two sequences is called arithmetic-harmonic mean since the arithmetic mean of a and b is the average $(a+b)/2$, and the harmonic mean is the inverse of the average inverse: $1/h = (1/a + 1/b)/2 = (a+b)/(2ab)$. The following program checks the limit for given numerical values:

```
sage: U = 2.0; V = 50.0
sage: while V-U >= 1.0e-6:      # 1.0e-6 stands for 1.0*10^-6
....:   temp = U
....:   U = 2 * U * V / (U + V)
....:   V = (temp + V) / 2
sage: U, V
(9.99999999989256, 10.0000000001074)
```

The values u_{n+1} and v_{n+1} depend from u_n and v_n; for this reason the main loop of this program introduces an auxiliary variable `temp` to correctly compute the new values u_{n+1}, v_{n+1} of `U`, `V` from the previous values u_n, v_n. The two left blocs below define the same sequences, while the right one builds two other sequences:

```
temp = 2*U*V/(U+V)
V = (U+V)/2
U = temp
```

```
U,V = 2*U*V/(U+V),(U+V)/2
```

(parallel assignment)

```
U = 2*U*V/(U+V)
V = (U+V)/2
```

$u'_{n+1} = \frac{2u'_n v'_n}{u'_n + v'_n}$

$v'_{n+1} = \frac{u'_{n+1} + v'_n}{2}$

The series $S_n = \sum_{k=0}^{n} (-1)^k a_k$ is *alternating* as soon as the sequence $(a_k)_{k\in\mathbb{N}}$ is decreasing and tends to zero. Since S is alternating, the two subsequences $(S_{2n})_{n\in\mathbb{N}}$ and $(S_{2n+1})_{n\in\mathbb{N}}$ satisfy Eq. (3.1), with common limit say ℓ. Hence the sequence $(S_n)_{n\in\mathbb{N}}$ also converges to ℓ and we have $S_{2p+1} \le \ell = \lim_{n\to+\infty} S_n \le S_{2p}$.

The following program illustrates this result for the sequence $a_k = 1/k^3$ from $k = 1$, by storing in two variables `U` and `V` the partial sums S_{2n} and S_{2n+1} surrounding the limit:

```
sage: U = 0.0        # the sum S0 is empty, of value zero
sage: V = -1.0       # S1 = -1/1^3
sage: n = 0          # U and V contain S(2n) and S(2n+1)
sage: while U-V >= 1.0e-6:
....:   n = n+1             # n += 1 is equivalent
....:   U = V + 1/(2*n)^3   # going from S(2n-1) to S(2n)
....:   V = U - 1/(2*n+1)^3 # going from S(2n) to S(2n+1)
sage: V, U
(-0.901543155458595, -0.901542184868447)
```

The main loop increases the value of n until the two terms S_{2n} and S_{2n+1} are close enough. The two variables `U` and `V` contain two consecutive terms; the loop body computes S_{2n} from S_{2n-1}, and then S_{2n+1} from S_{2n}, whence the crossed assignments to `U` and `V`.

The program halts when two consecutive terms S_{2n+1} and S_{2n} surrounding the limit are close enough, the approximation error — without taking into account rounding errors — satisfies then $0 \leq a_{2n+1} = S_{2n} - S_{2n+1} \leq 10^{-6}$.

Programming those five alternating series is similar:

$$\sum_{n\geq 2} \frac{(-1)^n}{\log n}, \qquad \sum_{n\geq 1} \frac{(-1)^n}{n}, \qquad \sum_{n\geq 1} \frac{(-1)^n}{n^2},$$
$$\sum_{n\geq 1} \frac{(-1)^n}{n^4}, \qquad \sum_{n\geq 1} (-1)^n e^{-n\,\ln n} = \sum_{n\geq 1} \frac{(-1)^n}{n^n}.$$

The terms of those series converge more or less rapidly to 0, thus the limit approximations require more or less computations.

Looking for a precision of 3, 10, 20 or 100 digits on the limits of those series consists in solving the following inequalities:

$$1/\log n \leq 10^{-3} \iff n \geq e^{(10^3)} \approx 1.97 \cdot 10^{434}$$

$$1/n \leq 10^{-3} \iff n \geq 10^3 \qquad 1/n \leq 10^{-10} \iff n \geq 10^{10}$$
$$1/n^2 \leq 10^{-3} \iff n \geq \sqrt{10^3} \approx 32 \qquad 1/n^2 \leq 10^{-10} \iff n \geq 10^5$$
$$1/n^4 \leq 10^{-3} \iff n \geq (10^3)^{1/4} \approx 6 \qquad 1/n^4 \leq 10^{-10} \iff n \geq 317$$
$$e^{-n\,\log n} \leq 10^{-3} \iff n \geq 5 \qquad e^{-n\,\log n} \leq 10^{-10} \iff n \geq 10$$

$$1/n^2 \leq 10^{-20} \iff n \geq 10^{10} \qquad 1/n^2 \leq 10^{-100} \iff n \geq 10^{50}$$
$$1/n^4 \leq 10^{-20} \iff n \geq 10^5 \qquad 1/n^4 \leq 10^{-100} \iff n \geq 10^{25}$$
$$e^{-n\,\log n} \leq 10^{-20} \iff n \geq 17 \qquad e^{-n\,\log n} \leq 10^{-100} \iff n \geq 57$$

In the simplest cases solving those inequalities yields an index n up from which the value S_n is close enough to the limit ℓ, and then a `for` enumeration loop is possible. However, when it is not possible to solve the inequality $a_n \leq 10^{-p}$, a `while` loop is necessary.

Numerical approximations of some of the above limits are too expensive, in particular when the index n gets as large as 10^{10} or 10^{12}. A mathematical study can sometimes determine the limit or approach it by other methods, like for Riemann series:

$$\lim_{n\to+\infty}\sum_{k=1}^{n}\frac{(-1)^k}{k^3}=-\frac{3}{4}\zeta(3),\qquad \text{with } \zeta(p)=\lim_{n\to+\infty}\sum_{k=1}^{n}\frac{1}{k^p},$$
$$\lim_{n\to+\infty}\sum_{k=1}^{n}\frac{(-1)^k}{k}=-\log 2,\qquad \lim_{n\to+\infty}\sum_{k=1}^{n}\frac{(-1)^k}{k^2}=-\frac{\pi^2}{12},$$
$$\lim_{n\to+\infty}\sum_{k=1}^{n}\frac{(-1)^k}{k^4}=-\frac{7\pi^4}{6!}.$$

Sage is able to compute symbolically some of these series, and determine a 1200-digit numerical approximation of $\zeta(3)$ in a few seconds, by doing far less operations than the 10^{400} ones required by the definition:

```
sage: k = var('k');                    sum((-1)^k/k, k, 1, +oo)
-log(2)
sage: sum((-1)^k/k^2, k, 1, +oo),  sum((-1)^k/k^3, k, 1, +oo)
(-1/12*pi^2, -3/4*zeta(3))
sage: -3/4 * zeta (N(3, digits = 1200))
-0.90154267736969571404980362113358749307373971925537416134\
4
20366650637865433973481763984190520700144360964936834644553\
9
56386899699900496241033229762790592512109045633721202005003\
9
...
01999549265288929706980408015180833590815343731070535991927\
1
79897015140616356032852450242460506051977442139028914505453\
8
90196121635914683781391659806428667225534381770353976017030\
6262
```

3.2.2 Conditionals

Another important instruction is the conditional (or test), which enables us to execute some instructions depending on the result of a boolean condition. The structure of the conditional and two possible syntaxes are:

```
if a condition:
    an instruction sequence
```

```
if a condition:
    an instruction sequence
else:
    another instruction sequence
```

The Syracuse sequence is defined using a parity condition:

$$u_0\in\mathbb{N}-\{0\},\qquad u_{n+1}=\begin{cases}u_n/2 & \text{if } u_n \text{ is even}\\ 3u_n+1 & \text{if } u_n \text{ is odd.}\end{cases}$$

The Collatz conjecture says — with no known proof in 2016 — that for all initial values $u_0 \in \mathbb{N} - \{0\}$, there exists a rank n for which $u_n = 1$. The next terms are then 4, 2, 1, 4, 2, etc. The computation of each term of this sequence requires a parity test. This condition is checked within a *while* loop, which determines the smallest $n \in \mathbb{N}$ satisfying $u_n = 1$:

```
sage: u = 6 ; n = 0
sage: while u != 1:    # the test u <> 1 is also possible
....:   if u % 2 == 0: # the operator % yields the remainder
....:     u = u//2     # //: Euclidean division quotient
....:   else:
....:     u = 3*u+1
....:   n = n+1
sage: n
8
```

Checking whether u_n is even is done by comparing to 0 the remainder of the Euclidean division of u_n by 2. The variable `n` at the end of the block is the number of iterations. The loop ends as soon as $u_n = 1$; for example if $u_0 = 6$ then $u_8 = 1$ and $8 = \min\{p \in \mathbb{N} | u_p = 1\}$:

$p =$	0	1	2	3	4	5	6	7	8	9	10	$\cdots$
$u_p =$	6	3	10	5	16	8	4	2	1	4	2	$\cdots$

Verifying step-by-step the correct behaviour of the loop can be done using a *spy-instruction* `print u, n` inside the loop body.

The `if` instruction also allows nested tests in the `else` branch using the `elif` keyword. The two following structures are thus equivalent:

```
if a condition cond1:                    if cond1:
     an instruction sequence inst1           inst1
else:                                    elif cond2:
    if a condition cond2:                    inst2
         an instruction sequence inst2   elif cond3:
    else:                                    inst3
        if a condition cond3:             else:
             an instruction sequence inst3       instn
        else:
             in other cases instn
```

Like for loops, small instruction sequences might be put after the colon on the same line, and not in an indented block below.

3.2.3 Procedures and Functions

General Syntax. As in other computer languages, the Sage user can define her/his own procedures or functions, using the `def` command whose syntax is detailed below. In this book, we call a *function* (resp. *procedure*) a sub-program

with zero, one or several arguments, which returns (resp. does not return) a result. Let us define the function $(x, y) \mapsto x^2 + y^2$:

```
sage: def fct2 (x, y):
....:   return x^2 + y^2
sage: a = var('a')
sage: fct2 (a, 2*a)
5*a^2
```

The function evaluation ends with the `return` command, whose argument, here $x^2 + y^2$, is the function result.

A procedure is like a function, but does not return any value, and without any `return` instruction the instruction body of the procedure is evaluated until its end. In fact a procedure returns the `None` value, which means "nothing".

By default, all variables appearing in a function are considered as local variables. Local variables are created at each function call, destroyed at the end of the function, and do not interact with other variables of the same name. In particular, global variables are not modified by the evaluation of a function having local variables of the same name:

```
sage: def foo (u):
....:   t = u^2
....:   return t*(t+1)
sage: t = 1 ; u = 2
sage: foo(3), t, u
(90, 1, 2)
```

It is possible to modify a global variable from within a function, with the `global` keyword:

```
sage: a = b = 1
sage: def f(): global a; a = b = 2
sage: f(); a, b
(2, 1)
```

Consider again the computation of the arithmetic-harmonic mean of two positive numbers:

```
sage: def AHmean (u, v):
....:     u, v = min(u, v), max(u, v)
....:     while v-u > 2.0e-8:
....:         u, v = 2*u*v/(u+v), (u+v)/2
....:     return (u+v) / 2
```

```
sage: AHmean (1., 2.)
1.41421356237309
sage: AHmean                        # corresponds to a function
  <function AHmean at ...>
```

The `AHmean` function has two parameters `u` and `v` which are local variables, whose initial values are those of the function arguments; for example with `AHmean (1., 2.)` the function body begins with $u = 1.0$ and $v = 2.0$.

The structured programming paradigm recommends to have the `return` statement at the very end of the function body. However, it is possible to put it in the middle of the instruction block, then the following instructions will not be executed. And the function body might contain several `return` occurrences.

Translating the mathematics spirit into the computer suggests to use functions which return results from their arguments, instead of procedures that output those results with a `print` command. The Sage computer algebra system is built itself on numerous functions like `exp` or `solve`, which return a result, for example a number, an expression, a list of solutions, etc.

Iterative and Recursive Methods. As we have seen above, a user-defined function is a sequence of instructions. A function is said *recursive* when during its evaluation, it calls itself with different parameters. The factorial sequence is a toy example of recursive sequence:

$$0! = 1, \qquad (n+1)! = (n+1)\,n! \quad \text{for all } n \in \mathbb{N}.$$

The two following functions yield the same result for a nonnegative integer argument n; the first function uses the iterative method with a `for` loop, while the second one is a *word-by-word* translation of the above recursive definition:

```
sage: def fact1 (n):
....:     res = 1
....:     for k in [1..n]: res = res*k
....:     return res
```

```
sage: def fact2 (n):
....:     if n == 0: return 1
....:     else: return n*fact2(n-1)
```

The Fibonacci sequence is a recurrent relation of order 2 since u_{n+2} depends on u_n and u_{n+1}:

$$u_0 = 0, \qquad u_1 = 1, \qquad u_{n+2} = u_{n+1} + u_n \quad \text{for all } n \in \mathbb{N}.$$

The function `fib1` below applies an iterative scheme to compute terms of the Fibonacci sequence: the variables `U` and `V` store the two previous values before computing the next one:

```
sage: def fib1 (n):
....:   if n == 0 or n == 1: return n
....:   else:
....:     U = 0 ; V = 1 # the initial terms u0 and u1
....:     for k in [2..n]: W = U+V ; U = V ; V = W
....:     return V
sage: fib1(8)
```

```
21
```

The `for` loop applies the relation $u_n = u_{n-1} + u_{n-2}$ from $n = 2$. Note: a parallel assignment `U,V = V,U+V` in place of `W=U+V ; U=V ; V=W` would avoid the need of an auxiliary variable `W`, and would translate the order-1 vectorial recurrence $X_{n+1} = f(X_n)$ with $f(a, b) = (b, a + b)$, for $X_n = (u_n, u_{n+1})$. Those iterative methods are efficient, however programming them requires to manually deal with variables corresponding to different terms of the sequence.

In contrary, the recursive function `fib2` follows more closely the mathematical definition of the Fibonacci sequence, which makes its programming and understanding easier:

```
sage: def fib2 (n):
....:   if 0 <= n <= 1: return n      # for n = 0 or n = 1
....:   else: return fib2(n-1) + fib2(n-2)
```

The result of this function is the value returned by the conditional statement: either 0 or 1 respectively for $n = 0$ and $n = 1$, otherwise the sum `fib2(n-1)+fib2(n-2)`; each branch of the test consists of a `return` instruction.

This method is however less efficient since several computations are duplicated. For example `fib2(5)` evaluates `fib2(3)` and `fib2(4)`, which are in turn evaluated in the same manner. Therefore, Sage computes twice `fib2(3)` and three times `fib2(2)`. This recursive process ends by the evaluation of either `fib2(0)` or `fib2(1)`, of value 0 or 1, and the evaluation of `fib2(n)` eventually consists in computing u_n by adding u_n ones, and u_{n-1} zeroes. The total number of additions performed to compute u_n is thus $u_{n+1} - 1$. This number grows very quickly, and no computer is able to compute u_{100} this way.

Other methods are also possible, for example remember the intermediate terms using the decorator `@cached_function`, or use properties of matrix powers: the following paragraph shows how to compute the millionth term of this sequence. For example, compare the efficiency of the function `fib2` defined above with the following one, for example on $n = 30$:

```
sage: @cached_function
sage: def fib2a (n):
....:   if 0 <= n <= 1: return n
....:   else: return fib2a(n-1) + fib2a(n-2)
```

3.2.4 Example: Fast Exponentiation

The naive method to compute a^n for $n \in \mathbb{N}$ performs n multiplications by a using a `for` loop:

```
sage: a = 2; n = 6; res = 1          # 1 is the product neutral element
sage: for k in [1..n]: res = res*a
sage: res                            # the value of res is 2^6
64
```

Integer powers often arise in mathematics and computer science; this paragraph discusses a general method to compute a^n much faster than the naive method. The sequence $(u_n)_{n\in\mathbb{N}}$ below satisfies $u_n = a^n$; this follows by induction from the equalities $a^{2k} = (a^k)^2$ and $a^{k+1} = a\,a^k$:

$$u_n = \begin{cases} 1 & \text{if } n = 0, \\ u_{n/2}^2 & \text{if } n \text{ is even positive,} \\ a\,u_{n-1} & \text{if } n \text{ is odd.} \end{cases} \tag{3.2}$$

For example, for $n = 11$:

$$u_{11} = a\,u_{10}, \quad u_{10} = u_5^2, \quad u_5 = a\,u_4, \quad u_4 = u_2^2,$$
$$u_2 = u_1^2, \quad u_1 = a\,u_0 = a;$$

therefore:

$$u_2 = a^2, \quad u_4 = u_2^2 = a^4, \quad u_5 = a\,a^4 = a^5,$$
$$u_{10} = u_5^2 = a^{10}, \quad u_{11} = a\,a^{10} = a^{11}.$$

The computation of u_n only involves terms u_k with $k \in \{0, ..., n-1\}$, and is thus well performed in a finite number of operations.

This example also shows that u_{11} is obtained after the evaluation of 6 terms u_{10}, u_5, u_4, u_2, u_1 and u_0, which performs 6 multiplications only. In general, the computation of u_n requires between $\log n/\log 2$ and $2\log n/\log 2$ multiplications. Indeed, u_n is obtained from u_k, $k \le n/2$, with one or two additional steps, according to the parity of n. This method is thus much faster than the naive one when n is large: about twenty products for $n = 10^4$ instead of 10^4 products:

indices k:	10 000	5 000	2 500	1 250	625	624	312	156	78
	39	38	19	18	9	8	4	2	1

However, this method is not always the best one; the following operations using b, c, d and f perform 5 products to compute a^{15}, whereas the above method — using u, v, w, x and y — requires 6 products, without counting the initial product $a \cdot 1$:

$b = a^2 \quad c = ab = a^3 \quad d = c^2 = a^6 \quad f = cd = a^9 \quad df = a^{15}$: 5 products;
$u = a^2 \quad v = au = a^3 \quad w = v^2 = a^6$
$x = aw = a^7 \quad y = x^2 = a^{14} \quad ay = a^{15}$: 6 products.

The recursive function `pow1` uses the recurrent sequence (3.2) to compute a^n:

```
sage: def pow1 (a, n):
....:   if n == 0: return 1
....:   elif n % 2 == 0: b = pow1 (a, n//2); return b*b
....:   else: return a * pow1(a, n-1)
```

```
sage: pow1 (2, 11)                # result is 2^11
```

```
2048
```

The number of operations performed by this function is the same as a computation by hand using (3.2). In the case n even, if the instructions `b = pow1(a, n//2);return b*b` would be replaced by `pow1(a, n//2)*pow1(a, n//2)`, Sage would perform much more computations because, like for the recursive function `fib2` for the Fibonacci sequence, some calculations would be duplicated. We would then have of the order of n products, i.e., as many as with the naive method.

Note that instead of `b = pow1(a, n//2);return b*b`, we could write `return pow1(a*a, n//2)`.

The program below performs the same computation of a^n using an iterative method:

```
sage: def pow2 (u, k):
....:   v = 1
....:   while k != 0:
....:     if k % 2 == 0: u = u*u ; k = k//2
....:     else: v = v*u ; k = k-1
....:   return v
```

```
sage: pow2 (2, 10)                # result is 2^10
1024
```

The fact that `pow2(a, n)` returns a^n is shown by verifying that after each iteration the values of the variables `u`, `v` and `k` satisfy $v\,u^k = a^n$, for whatever parity of k. Before the first iteration $v = 1$, $u = a$ and $k = n$; after the last one $k = 0$, thus $v = a^n$.

The successive values of the integer variable `k` are nonnegative, and they form a decreasing sequence. Hence this variable can only take a finite number of values before being zero and terminating the loop.

Despite their apparent differences — `pow1` is recursive, while `pow2` is iterative — those two functions express almost the same algorithm: the only difference is that a^{2k} is evaluated as $(a^k)^2$ in `pow1`, and as $(a^2)^k$ in `pow2`, through the update of the variable `u`.

The method presented here is not limited to the computation of a^n where a is a number and n a positive integer, it applies to any associative law (which is needed to preserve usual properties of iterated products). For instance, by replacing the integer 1 by the $m \times m$ unit matrix $\mathbf{1}_m$, the two above functions would evaluate powers of square matrices. Those functions show how to efficiently implement the power operator "^" upon multiplication, and are similar to the method implemented within Sage.

For example, using powers of matrices enables us to compute much larger terms of the Fibonacci sequence:

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}, \qquad X_n = \begin{pmatrix} u_n \\ u_{n+1} \end{pmatrix}, \qquad AX_n = X_{n+1}, \qquad A^n X_0 = X_n.$$

The corresponding Sage program fits in two lines, and the wanted result is the first coordinate of the matrix product $A^n X_0$, which effectively works for $n = 10^7$; the `fib3` and `fib4` programs are equivalent, and their efficiency comes from the fact that Sage implements a fast exponentiation method:

```
sage: def fib3 (n):
....:   A = matrix ([[0, 1], [1, 1]]) ; X0 = vector ([0, 1])
....:   return (A^n*X0)[0]
```

```
sage: def fib4 (n):
....:   return (matrix([[0,1], [1,1]])^n * vector([0,1]))[0]
```

3.2.5 Input and Output

The `print` instruction is the main output command. By default, its arguments are printed one after the other, separated by spaces, with a newline after the command:

```
sage: print 2^2, 3^3, 4^4 ; print 5^5, 6^6
4 27 256
3125 46656
```

A comma at the end tells the next `print` instruction to continue on the same line:

```
sage: for k in [1..10]: print '+', k,
+ 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
```

To print results without intermediate spaces, we can transform them into a character string using the `str(..)` function, and concatenate strings with the "+" operator:

```
sage: print 10, 0.5 ; print 10+0.5 ; print 10.0, 5
10 0.500000000000000
10.5000000000000
10.0000000000000 5
sage: print 10+0, 5 ; print str(10)+str(0.5)
10 5
100.500000000000000
```

The last section of this chapter discusses in more detail character strings.

The `print` command is also able to format the output: the following example prints a table of fourth powers using the `%.d` placeholder and the `%` operator:

```
sage: for k in [1..6]: print '%2d^4 = %4d' % (k, k^4)
 1^4 =    1
 2^4 =   16
 3^4 =   81
 4^4 =  256
 5^4 =  625
 6^4 = 1296
```

The `%` operator replaces the expressions to its right in the character string to its left, in place of the placeholders like `%2d` or `%.4f`. In the above example the `%4d` specifier adds some left padding spaces to the string representing k^4, to get at least four characters. Likewise, the `%.4f` placeholder in `'pi = %.4f' % n(pi)` outputs `pi = 3.1416` with four digits after the decimal point.

In a terminal, the `raw_input('message')` command prints the text `message`, waits a keyboard input validated by the ⟨Enter⟩ key, and returns the user-given character string.

3.3 Lists and Other Data Structures

This section discusses some data structures available in Sage: character strings, lists — either mutable or non-mutable —, sets and dictionaries.

3.3.1 List Creation and Access

The list in computer science and the n-tuple in mathematics allow the enumeration of mathematical objects. In a pair — with $(a, b) \neq (b, a)$ — and an n-tuple, each object has its own position, contrary to a set.

A list is defined by surrounding its elements with square brackets `[...]`, separated by commas. Assigning the triplet $(10, 20, 30)$ to the variable `L` is done as follows, and the empty list is defined as:

```
sage: L = [10, 20, 30]
sage: L
[10, 20, 30]
sage: []                    # [] is the empty list
[]
```

The list indices are increasing up from 0, 1, 2, etc. The element of index k of a list L is accessed simply by `L[k]`, in mathematical terms this corresponds to the canonical projection on the k-th coordinate. The number of elements of a list is given by the `len` function[5]:

```
sage: L[1], len(L), len([])
(20, 3, 0)
```

Modifying an element is done the same way, by simply assigning the corresponding index. Hence the following command modifies the third term of the list, whose index is 2:

```
sage: L[2] = 33
sage: L
[10, 20, 33]
```

Negative indices access to end-of-list elements, `L[-1]` refering to the last one:

[5]The output of `len` is a Python integer of type `int`, to get a Sage integer we write `Integer(len(..))`.

```
sage: L = [11, 22, 33]
sage: L[-1], L[-2], L[-3]
(33, 22, 11)
```

The command `L[p:q]` extracts the sub-list `[L[p], L[p+1], ..., L[q-1]]`, which is empty if $q \leq p$. Negative indices allow to reference the last terms of the list; finally `L[p:]` is equivalent to `L[p:len(L)]`, and `L[:q]` to `L[0:q]`:

```
sage: L = [0, 11, 22, 33, 44, 55]
sage: L[2:4]
[22, 33]
sage: L[-4:4]
[22, 33]
sage: L[2:-2]
[22, 33]
sage: L[:4]
[0, 11, 22, 33]
sage: L[4:]
[44, 55]
```

Similarly to the `L[n] = ...` command which modifies an element of the list, the assignment `L[p:q] = [...]` substitutes all elements between index p included and index q excluded:

```
sage: L = [0, 11, 22, 33, 44, 55, 66, 77]
sage: L[2:6] = [12, 13, 14]        # substitutes [22, 33, 44, 55]
```

Therefore `L[:1] = []` and `L[-1:] = []` delete respectively the first and last term of a list, and likewise `L[:0] = [a]` and `L[len(L):] = [a]` insert the element a respectively in front and in tail of the list. More generally the following equalities hold:

$$\texttt{L} = [\ell_0, \ell_1, \ell_2, \ldots, \ell_{n-1}] = [\ell_{-n}, \ell_{1-n}, \ldots \ell_{-2}, \ell_{-1}] \qquad \text{with } n = \texttt{len(L)},$$
$$\ell_k = \ell_{k-n} \quad \text{for } 0 \leq k < n, \qquad \ell_j = \ell_{n+j} \quad \text{for } -n \leq j < 0.$$

The operator `in` checks whether a list contains a given element, while "`==`" compares two lists elementwise. The two sub-lists below with positive or negative indices are equal:

```
sage: L = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
sage: L[3:len(L)-5] == L[3-len(L):-5]
True
sage: [5 in L, 6 in L]
[True, False]
```

While we have considered so far lists with integer elements, list elements can be any Sage object: numbers, expressions, other lists, etc.

3.3.2 Global List Operations

The addition operator "`+`" concatenates two lists, and the multiplication operator "`*`", together with an integer, performs an iterated concatenation:

```
sage: L = [1, 2, 3] ; L + [10, 20, 30]
[1, 2, 3, 10, 20, 30]
sage: 4 * [1, 2, 3]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The concatenation of the two sub-lists `L[:k]` and `L[k:]` reconstructs the original list. This explains why the left bound `p` of a sub-list `L[p:q]` is included, while the right bound `q` is excluded:

$$\begin{aligned} \texttt{L} &= \texttt{L[:k]+L[k:]} = [\ell_0, \ell_1, \ell_2, \ldots, \ell_{n-1}] \\ &= [\ell_0, \ell_1, \ell_2, \ldots, \ell_{k-1}] + [\ell_k, \ell_{k+1}, \ell_{k+2}, \ldots, \ell_{n-1}]. \end{aligned}$$

This property is shown in the following example:

```
sage: L = 5*[10, 20, 30] ; L[:3]+L[3:] == L
True
```

The operator made from two points "`..`" makes it easy to construct integer lists without explicitly enumerating all elements, and can be mixed with isolated elements:

```
sage: [1..3, 7, 10..13]
[1, 2, 3, 7, 10, 11, 12, 13]
```

We explain below how to build the image of a list by a function, and a sub-list of a list. The corresponding functions are `map` and `filter`, together with the `[..for..x..in..]` construction. Mathematics often involve lists made by applying a function f to its elements:

$$(a_0, a_1, \ldots, a_{n-1}) \mapsto (f(a_0), f(a_1), \ldots, f(a_{n-1})).$$

The `map` command builds this "map": the following example applies the trigonometric function cos to a list of usual angles:

```
sage: map (cos, [0, pi/6, pi/4, pi/3, pi/2])
[1, 1/2*sqrt(3), 1/2*sqrt(2), 1/2, 0]
```

A user-defined function — with `def` — or a lambda-expression might also be used as first argument of `map`; the following command is equivalent to the above, using the function $t \mapsto \cos t$:

```
sage: map (lambda t: cos(t), [0, pi/6, pi/4, pi/3, pi/2])
[1, 1/2*sqrt(3), 1/2*sqrt(2), 1/2, 0]
```

The `lambda` command is followed by the parameters separared by commas, and must have after the colon exactly one expression, which is the function result (without the `return` keyword).

A `lambda` expression might contain a test, whence the following functions are equivalent:

```
fctTest1 = lambda x:  res1 if cond else res2
def fctTest2 (x):
  if cond:  return res1
  else:  return res2
```

As a consequence, the three following `map` commands are equivalent, the composition $N \circ \cos$ being expressed in different ways:

```
sage: map (lambda t: N(cos(t)), [0, pi/6, pi/4, pi/3, pi/2])
[1.00000000000000, 0.866025403784439, 0.707106781186548,
0.500000000000000, 0.000000000000000]
```

```
sage: map (N, map (cos, [0, pi/6, pi/4, pi/3, pi/2]))
[1.00000000000000, 0.866025403784439, 0.707106781186548,
0.500000000000000, 0.000000000000000]
```

```
sage: map (compose(N, cos), [0, pi/6, pi/4, pi/3, pi/2])
[1.00000000000000, 0.866025403784439, 0.707106781186548,
0.500000000000000, 0.000000000000000]
```

The `filter` command builds the sub-list of the elements satisfying a given condition. To get all integers in $1, ..., 55$ that are prime:

```
sage: filter (is_prime, [1..55])
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
```

The test condition might be defined inside the `filter` command, as in the following example which finds by exhaustive search all fourth roots of 7 modulo the prime 37; this equation has four solutions 3, 18, 19 and 34:

```
sage: p = 37 ; filter (lambda n: n^4 % p == 7, [0..p-1])
[3, 18, 19, 34]
```

Another way to build a list is using the *comprehension* form `[..for..x..in..]`; both commands below enumerate odd integers from 1 to 31:

```
sage: map(lambda n:2*n+1, [0..15])
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]
sage: [2*n+1 for n in [0..15]]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]
```

The comprehension command is independent from the `for` loop. Associated with the `if` condition, it yields an equivalent construction to `filter`:

```
sage: filter (is_prime, [1..55])
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
sage: [p for p in [1..55] if is_prime(p)]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
```

In the two following examples, we combine the `if` and `filter` tests with the comprehension `for` to determine a list of primes congruent to 1 modulo 4, then a list of squares of prime numbers:

```
sage: filter (is_prime, [4*n+1 for n in [0..20]])
[5, 13, 17, 29, 37, 41, 53, 61, 73]
sage: [n^2 for n in [1..20] if is_prime(n)]
[4, 9, 25, 49, 121, 169, 289, 361]
```

In the first case the `is_prime` test is performed after the computation of $4n + 1$, while in the second one the primality test is done before the computation of the square n^2.

The `reduce` function operates by associativity from left to right on the elements of a list. Let us define the following operation, say $\star$:

$$x \star y = 10x + y, \qquad \text{then } ((1 \star 2) \star 3) \star 4 = (12 \star 3) \star 4 = 1234.$$

The first argument of `reduce` is a two-parameter function, the second one is the list of its arguments:

```
sage: reduce (lambda x, y: 10*x+y, [1, 2, 3, 4])
1234
```

A third optional argument gives the image of an empty list:

```
sage: reduce (lambda x, y: 10*x+y, [9, 8, 7, 6], 1)
19876
```

This third argument usually corresponds to the neutral element of the operation which is applied. The following example computes a product of odd integers:

```
sage: L = [2*n+1 for n in [0..9]]
sage: reduce (lambda x, y: x*y, L, 1)
654729075
```

The Sage functions `add`[6] and `prod` apply directly the `reduce` operator to compute sums and products; the three examples below yield the same result. The list form enables us to add an optional second argument which stands for the neutral element, 1 for the product and 0 for the sum, or a unit matrix for a matrix product:

```
sage: prod ([2*n+1 for n in [0..9]], 1) # a list with for
654729075
sage: prod ( 2*n+1 for n in [0..9])     # without a list
654729075
sage: prod (n for n in [0..19] if n%2 == 1)
654729075
```

The function `any` associated to the `or` operator, and the function `all` to the `and` operator, have similar syntax. Their evaluation terminates as soon as the result `True` or `False` obtained for one term avoids the evaluation of the next terms:

[6] Do not mix `add` with `sum`, which looks for a symbolic expression of a sum.

```
sage: def fct (x): return 4/x == 2
sage: all (fct(x) for x in [2, 1, 0])
False
sage: any (fct(x) for x in [2, 1, 0])
True
```

In contrast, the construction of the list `[fct(x) for x in [2, 1, 0]]` and the command `all([fct(x) for x in [2, 1, 0]])` produce an error because all terms are evaluated, including the last one with $x = 0$.

Nesting several `for` operators enables us to construct the cartesian product of two lists, or to define lists of lists. As seen in the following example, the leftmost `for` operator corresponds to the outermost loop:

```
sage: [[x, y] for x in [1..2] for y in [6..8]]
[[1, 6], [1, 7], [1, 8], [2, 6], [2, 7], [2, 8]]
```

The order whence differs from what is obtained by constructing a list of lists using nested `for` comprehensions:

```
sage: [[[x, y] for x in [1..2]] for y in [6..8]]
[[[1, 6], [2, 6]], [[1, 7], [2, 7]], [[1, 8], [2, 8]]]
```

The `map` command with several lists as arguments takes one element of each list in turn:

```
sage: map (lambda x, y: [x, y], [1..3], [6..8])
[[1, 6], [2, 7], [3, 8]]
```

Finally with the `flatten` command, we can concatenate lists on one or several levels:

```
sage: L = [[1, 2, [3]], [4, [5, 6]], [7, [8, [9]]]]
sage: flatten (L, max_level = 1)
[1, 2, [3], 4, [5, 6], 7, [8, [9]]]
sage: flatten (L, max_level = 2)
[1, 2, 3, 4, 5, 6, 7, 8, [9]]
sage: flatten (L)             # equivalent to flatten (L, max_level = 3)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Those elementary list operations are quite useful in other parts of Sage; the following example computes the first iterative derivatives of $x\,e^x$; the first argument of `diff` is the expression to differentiate, and the following argument is the derivation variable, or in the case of several arguments the variables with respect to which the expression should be successively differentiated:

```
sage: x = var('x')
sage: factor(diff(x*exp(x), [x, x]))
(x + 2)*e^x
sage: map(lambda n: factor(diff(x*exp(x), n*[x])), [0..6])
[x*e^x, (x + 1)*e^x, (x + 2)*e^x, (x + 3)*e^x, (x + 4)*e^x,
(x + 5)*e^x, (x + 6)*e^x]
sage: [factor (diff (x*exp(x), n*[x])) for n in [0..6]]
```

```
[x*e^x, (x + 1)*e^x, (x + 2)*e^x, (x + 3)*e^x, (x + 4)*e^x,
(x + 5)*e^x, (x + 6)*e^x]
```

The `diff` command admits more than one syntax. The parameters after the function f can be a list of variables, an enumeration of variables, or a variable and an order of derivation:

```
diff(f(x), x, x, x),   diff(f(x), [x, x, x]),   diff(f(x), x, 3).
```

We can also use `diff(f(x), 3)` for functions of one variable. The above results are a direct consequence of Leibniz' formula for iterated derivatives of a 2-term product, given the fact that the derivatives of order 2 or more of x are zero:

$$(xe^x)^{(n)} = \sum_{k=0}^{n} \binom{n}{k} x^{(k)} (e^x)^{(n-k)} = (x+n)e^x.$$

3.3.3 Main Methods on Lists

The `reverse` method reverts the order of elements in a list, and the `sort` method transforms the given list in a sorted one:

```
sage: L = [1, 8, 5, 2, 9] ; L.reverse() ; L
[9, 2, 5, 8, 1]
sage: L.sort() ; L
[1, 2, 5, 8, 9]
sage: L.sort(reverse = True) ; L
[9, 8, 5, 2, 1]
```

Both methods modify the list `L` in place, the initial list being lost.

A first optional argument of `sort` enables us to choose the order relation, in form of a two-parameter function `Order(x, y)`. The returned value of this function must have the type `int` of the Python integers; it is negative, zero or positive, for example -1, 0 or 1, when $x \prec y$, $x = y$ or $x \succ y$, respectively. The transformed list $(x_0, x_1, \ldots, x_{n-1})$ satisfies $x_0 \preceq x_1 \preceq \cdots \preceq x_{n-1}$.

The lexicographic order of two number lists of same length is similar to the alphabetic order and is defined as follows, ignoring the first equal terms:

$$\begin{aligned} & P = (p_0, p_1, \ldots p_{n-1}) \prec_{\text{lex}} Q = (q_0, q_1, \ldots q_{n-1}) \\ \iff & \exists r \in \{0, \ldots, n-1\} \quad (p_0, p_1, \ldots, p_{r-1}) = (q_0, q_1, \ldots, q_{r-1}) \text{ and } p_r < q_r. \end{aligned}$$

The following function compares two lists of equal lengths. Despite the *a priori* infinite loop `while True`, the `return` commands ensure the termination, together with the finite length. The result is -1, 0 or 1 according to $P \prec_{\text{lex}} Q$, $P = Q$ or $P \succ_{\text{lex}} Q$:

```
sage: def alpha (P, Q):     # len(P) = len(Q) by hypothesis
....:   i = 0
....:   while True:
....:     if i == len(P): return int(0)
....:     elif P[i] < Q[i]: return int(-1)
....:     elif P[i] > Q[i]: return int(1)
....:     else: i = i+1
```

```
sage: alpha ([2, 3, 4, 6, 5], [2, 3, 4, 5, 6])
1
```

The following command sorts a list of lists of same length using the lexicographic order. The `alpha` function using the same order as used by Sage to compare two lists, the command `L.sort()` without optinal argument is thus equivalent:

```
sage: L = [[2, 2, 5], [2, 3, 4], [3, 2, 4], [3, 3, 3],\
....: [1, 1, 2], [1, 2, 7]]
sage: L.sort (cmp = alpha) ; L
[[1, 1, 2], [1, 2, 7], [2, 2, 5], [2, 3, 4], [3, 2, 4], [3, 3, 3]]
```

The homogeneous lexicographic order first compares terms according to their weight, where the weight is the sum of coefficients, and for equal weights only resorts to the lexicographic order:

$$\begin{aligned} & P = (p_0, p_1, \dots p_{n-1}) \prec_{\text{lexH}} Q = (q_0, q_1, \dots q_{n-1}) \\ \iff & \sum_{k=0}^{n-1} p_k < \sum_{k=0}^{n-1} q_k \text{ or } \Big(\sum_{k=0}^{n-1} p_k = \sum_{k=0}^{n-1} q_k \text{ and } P \prec_{\text{lex}} Q \Big). \end{aligned}$$

This function implements the homogeneous lexicographic order:

```
sage: def homogLex (P, Q):
....:  sp = sum (P) ; sq = sum (Q)
....:  if sp < sq: return int(-1)
....:  elif sp > sq: return int(1)
....:  else: return alpha (P, Q)
```

```
sage: homogLex ([2, 3, 4, 6, 4], [2, 3, 4, 5, 6])
-1
```

The Sage function `sorted` is a function in the mathematical sense: it takes as first argument a list and returns the corresponding sorted list, without modifying its argument, unlike `sort`.

Sage provides other methods on lists, to insert an element at the tail, to append a list at the end, to count the number of occurrences of an element:

`L.append(x)`	is equivalent to `L[len(L):] = [x]`
`L.extend(L1)`	is equivalent to `L[len(L):] = L1`
`L.insert(i, x)`	is equivalent to `L[i:i] = [x]`
`L.count(x)`	is equivalent to `len (select (lambda t :  t == x, L))`

The commands `L.pop(i)` and `L.pop()` remove the element of index i, or the last one, and return the removed element; their behaviour is described by those two functions:

```
def pop1 (L, i):
  a = L[i]
  L[i:i+1] = []
  return a
```

```
def pop2 (L):
  return pop1 (L, len(L)-1)
```

In addition, `L.index(x)` returns the index of the first element equal to x, and `L.remove(x)` removes the first element equal to x. Those commands raise an error when x is not in the list. Finally, the command `del L[p:q]` is equivalent to `L[p:q] = []`, and `del L[i]` removes the ith element.

Contrary to what happens in several other computer languages, those functions modify in place the list `L`, without creating a new list.

3.3.4 Examples of List Manipulations

The following example constructs the list of even terms and the list of odd terms of a given list. This first solution goes twice through the list, and thus performs twice the parity tests:

```
sage: def fct1(L):
....:   return [filter (lambda n: n % 2 == 0, L),
....:           filter (lambda n: n % 2 == 1, L)]
```

```
sage: fct1([1..10])
[[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]]
```

The second function below goes only once through the list, and constructs the two result lists element by element:

```
sage: def fct2 (L):
....:   res0 = [] ; res1 = []
....:   for k in L:
....:     if k%2 == 0: res0.append(k) # or res0[len(res0):] = [k]
....:     else: res1.append(k)        # or res1[len(res1):] = [k]
....:   return [res0, res1]
```

This program replaces the `for` loop and the auxiliary variables by a recursive call and an additional parameter:

```
sage: def fct3a (L, res0, res1):
....:   if L == []: return [res0, res1]
....:   elif L[0]%2 == 0: return fct3a(L[1:], res0+[L[0]], res1)
....:   else: return fct3a (L[1:], res0, res1+[L[0]])
```

```
sage: def fct3 (L): return fct3a (L, [], [])
```

The parameters `res0` and `res1` contain the first element already treated, and the parameter list `L` has one term less at each recursive call.

The second example below extracts all maximal non-decreasing sequences of a list of numbers. Three variables are used, the first one `res` keeps track of all non-decreasing sequences already obtained, the `start` variable is the starting index of the current sub-sequence, and `k` is the loop index:

```
sage: def subSequences (L):
....:   if L == []: return []
....:   res = [] ; start = 0 ; k = 1
```

```
....:   while k < len(L):    # 2 consecutive terms are defined
....:     if L[k-1] > L[k]:
....:       res.append (L[start:k]) ; start = k
....:     k = k+1
....:   res.append (L[start:k])
....:   return res
```

```
sage: subSequences([1, 4, 1, 5])
[[1, 4], [1, 5]]
sage: subSequences([4, 1, 5, 1])
[[4], [1, 5], [1]]
```

The loop body deals with the kth element of the list. If the condition is fulfilled, the current non-decreasing sub-sequence ends, and we start a new sub-sequence, otherwise the current sub-sequence is extended by one term.

After the loop body, the `append` instruction adds to the final result the current sub-sequence, which contains at least one element.

3.3.5 Character Strings

Character strings are delimited by single or double quotes, `'...'` or `"..."`. Strings delimited by single quotes might contain double quotes, and vice versa. Strings can also be delimited by triple quotes `'''...'''`: in that case they might span several lines and contain single or double quotes.

```
sage: S = 'This is a character string.'
```

The escape character is the \ symbol, which allows to include end of lines by `\n`, quotes by `\"` or `\'`, tabulations by `\t`, the backslash character by `\\`. Character strings might contain characters with accents, and more generally any Unicode character:

```
sage: S = 'This is a déjà-vu example.'; S
'This is a d\xc3\xa9j\xc3\xa0-vu example.'
sage: print S
This is a déjà-vu example.
```

The comparison of two character strings is performed according to the internal encoding of each character. The length of a string is given by the `len` function, and the concatenation of strings is performed by the addition and multiplication symbols "`+`" and "`*`".

Accessing to sub-strings of `S` is done like for lists using square brackets `S[n]`, `S[p:q]`, `S[p:]` and `S[:q]`, the result being a character string. The language forbids to replace an initial string by such an assignment, for this reason character strings are *non-mutable*.

The `str` function converts its argument into a character string. The `split` method cuts a given string at spaces:

```
sage: S='one two three four five six seven'; L=S.split(); L
['one', 'two', 'three', 'four', 'five', 'six', 'seven']
```

The Python very extensive `re` library might also be used to search sub-strings, words and regular expressions.

3.3.6 Shared or Duplicated Data Structures

A list in square brackets `[...]` can be modified by assigning some of its elements, by a change of the number of elements, or by methods like `sort` or `reverse`.

Assigning a list to a variable does not duplicate the data structure, which is shared. In the following example the lists `L1` and `L2` remain identical: they correspond to two *aliases* of the same object, and modifying one of them is visible on the other one:

```
sage: L1 = [11, 22, 33] ; L2 = L1
sage: L1[1] = 222 ; L2.sort() ; L1, L2
([11, 33, 222], [11, 33, 222])
sage: L1[2:3] = []; L2[0:0] = [6, 7, 8]
sage: L1, L2
([6, 7, 8, 11, 33], [6, 7, 8, 11, 33])
```

In contrast, the `map`, `filter` and `flatten` functions duplicate the data structures, as well as the list construction by `L[p:q]` or `[..for..if..]`, the concatenation by `+` and `*`.

In the above example, replacing on the first line `L2 = L1` by one of the next six commands completely changes the following results, since modifications on one list do not propagate on the other one. The two structures become independent, the two lists are distinct even if they have the same value; for example the assignment `L2 = L1[:]` copies the sub-list of `L1` from the first to last term, and thus fully duplicates the structure of `L1`:

```
L2 = [11, 22, 33]  L2 = copy(L1)  L2 = L1[:]
L2 = []+L1         L2 = L1+[]     L2 = 1*L1
```

Checking for shared data structures can be done in Sage using the `is` binary operator; if the answer is true, all modifications will have a side effect on both variables:

```
sage: L1 = [11, 22, 33] ; L2 = L1 ; L3 = L1[:]
sage: [L1 is L2, L2 is L1, L1 is L3, L1 == L3]
[True, True, False, True]
```

Copy operations on lists operate on one level only. As a consequence, modifying an element in a list of lists has a side effect despite the list copy at the outer level:

```
sage: La = [1, 2, 3] ; L1 = [1, La] ; L2 = copy(L1)
sage: L1[1][0] = 5         # [1, [5, 2, 3]] for L1 and L2
sage: [L1 == L2, L1 is L2, L1[1] is L2[1]]
[True, False, True]
```

The following instruction duplicates a list on two levels:

```
sage: map (copy, L)
```

whereas the `copyRec` function recursively duplicates a list at all levels:

```
sage: def copyRec (L):
....:  if type (L) == list: return map (copyRec, L)
....:  else: return L
```

The inverse lexicographic order is defined from the lexicographic order on n-tuples by reversing the order on each element:

$$\begin{aligned} & P = (p_0, p_1, \dots p_{n-1}) \prec_{\text{lexInv}} Q = (q_0, q_1, \dots q_{n-1}) \\ \iff & \exists r \in \{0, \dots, n-1\}, \quad (p_{r+1}, \dots, p_{n-1}) = (q_{r+1}, \dots, q_{n-1}) \text{ and } p_r > q_r. \end{aligned}$$

Programming this inverse lexicographic order might be done using the above-defined `alpha` function, which implements the lexicographic order. We have to copy the lists P and Q to perform the inversion without modifying those lists. More precisely the `lexInverse` function reverts the n-tuples by `reverse`, and returns the opposite of the Python integer corresponding to the wanted comparison: $-(P_1 \prec_{\text{lex}} Q_1)$:

```
sage: def lexInverse (P, Q):
....:   P1 = copy(P) ; P1.reverse()
....:   Q1 = copy(Q) ; Q1.reverse()
....:   return - alpha (P1, Q1)
```

The changes made on a list given as argument of a function are performed on the original list, since the functions do not copy arguments which are lists. Thus a function that would perform `P.reverse()`, in place of `P1 = copy(P)` and `P1.reverse()`, would modify definitively the list P; this *side effect* is usually not wanted.

The variable P is a local variable of the function, independent from any other global variable also called P, but this has nothing to do with modifications made to a list given as argument of the function.

The lists in Python and Sage are implemented as dynamic tables, contrary to Lisp and OCaml where lists are defined by a head t and a tail list Q. The Lisp command `cons(t,Q)` returns a list with head t without modifying the list Q, whereas in Python, adding an element e to a dynamic table T via `T.append(e)` modifies the table T. Both representations have advantages and drawbacks, and switching from one to the other one is possible, however the efficiency of a given algorithm might greatly vary from one representation to the other one.

3.3.7 Mutable and Non-mutable Data Structures

List enable us to construct and manipulate elements that can be modified: they are called *mutable* data structures.

Python also allows to define non-mutable objects. The non-mutable data structure corresponding to lists is called *sequence* or *tuple*, and is denoted with parentheses `(...)` instead of square brackets `[...]`. A tuple with only one element is defined by adding a comma after this element, to distinguish it from mathematical parentheses.

```
sage: S0 = (); S1 = (1, ); S2 = (1, 2)
sage: [1 in S1, 1 == (1)]
[True, True]
```

The operations on tuples are essentially the same as those on lists, for example `map` constructs the image of a tuple by a function, `filter` extracts a sub-sequence. In all cases the result is a list, and the `for` comprehension transforms a tuple in list:

```
sage: S1 = (1, 4, 9, 16, 25); [k for k in S1]
[1, 4, 9, 16, 25]
```

The `zip` command groups term-by-term several lists or tuples, and is equivalent to the following `map` command:

```
sage: L1 = [0..4]; L2 = [5..9]
sage: zip(L1, L2)
[(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]
sage: map(lambda x, y:(x, y), L1, L2)
[(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]
```

3.3.8 Finite Sets

Contrary to lists, the set data structure only accounts whether an element is present or absent, without considering any position or number of repetitions. Sage constructs finite sets from the `Set` function, applied to the list of its elements. The result is output with curly brackets:

```
sage: E = Set([1, 2, 4, 8, 2, 2, 2]); F = Set([7, 5, 3, 1]); E, F
({8, 1, 2, 4}, {1, 3, 5, 7})
```

The operator `in` checks whether a set contains a given element, and Sage allows the union of sets by `+` or `|`, the intersection by `&`, the set difference by `-`, and the symmetric difference using `^^`:

```
sage: E = Set([1, 2, 4, 8, 2, 2, 2]); F = Set([7, 5, 3, 1])
sage: 5 in E, 5 in F, E + F == F | E
(False, True, True)
sage: E & F, E - F, E ^^ F
({1}, {8, 2, 4}, {2, 3, 4, 5, 7, 8})
```

The `len(E)` command gives the cardinality of such a finite set. The operations `map`, `filter` and `for..if...` apply to sets as well as tuples, and yield lists as results. The access to a given element is done via `E[k]`. The commands below construct in two different ways the list of elements of a set:

```
sage: E = Set([1, 2, 4, 8, 2, 2, 2])
sage: [E[k] for k in [0..len(E)-1]], [t for t in E]
([8, 1, 2, 4], [8, 1, 2, 4])
```

The following function checks whether E is a subset of F, using the union operator:

```
sage: def included (E, F): return E+F == F
```

Contrary to lists, sets are non-mutable, and thus cannot be modified; their elements must also be non-mutable. Sets of tuples or sets of sets are thus possible, but not sets of lists:

```
sage: Set([Set([]), Set([1]), Set([2]), Set([1, 2])])
{{1, 2}, {}, {2}, {1}}
sage: Set([    (),     (1, ),    (2, ),    (1, 2) ])
{(1, 2), (2,), (), (1,)}
```

The following function scans all the subsets of a set via a recursive method:

```
sage: def Parts (EE):
....:   if EE == Set([]): return Set([EE])
....:   else:
....:     return withOrWithout (EE[0], Parts(Set(EE[1:])))
```

```
sage: def withOrWithout (a, E):
....:   return Set (map (lambda F: Set([a])+F, E)) + E
```

```
sage: Parts(Set([1, 2, 3]))
{{3}, {1, 2}, {}, {2, 3}, {1}, {1, 3}, {1, 2, 3}, {2}}
```

The `withOrWithout(a, E)` function call takes a set E of subsets, and constructs the set twice as large made from those subsets, and those subsets added (in the set union sense) with a. The recursive construction starts with a set with one element $E = \{\emptyset\}$.

3.3.9 Dictionaries

Last but not least, Python, and thus Sage, provides the notion of dictionary. Like a phone book, a dictionary associates a value to a given key.

The keys of a dictionary might be of any non-mutable type, numbers, characters strings, tuples, etc. The syntax is like lists, using assignments from the empty dictionary `dict()` which can be written `{}` too:

```
sage: D={}; D['one']=1; D['two']=2; D['three']=3; D['ten']=10
sage: D['two'] + D['three']
5
```

The above example shows how to add an entry (key,value) in a dictionary, and how to access the value associated to a given key via `D[...]`.

The operator `in` checks whether a key is in a dictionary, and the commands `del D[x]` or `D.pop(x)` erase the entry of key x in this dictionary.

The following example demonstrates how a dictionary can be used to represent a function on a finite set:

$$E = \{a_0, a_1, a_2, a_3, a_4, a_5\}, \quad \begin{array}{lll} f(a_0) = b_0, & f(a_1) = b_1, & f(a_2) = b_2, \\ f(a_3) = b_0, & f(a_4) = b_3, & f(a_5) = b_3. \end{array}$$

Methods on dictionaries are comparable to those on other enumerated data structures. The program below implements the above function, and gives the input set E and the output set $\mathrm{Im}\, f = f(E)$ via the methods `keys` and `values`:

```
sage: D = {'a0':'b0', 'a1':'b1', 'a2':'b2', 'a3':'b0',\
....: 'a4':'b3', 'a5':'b3'}
sage: E  = Set(D.keys()) ; Imf = Set(D.values())
sage: Imf == Set(map (lambda t:D[t], E))     # is equivalent
True
```

This last command directly translates the mathematical definition $\mathrm{Im}\, f = \{f(x)|x \in E\}$. Dictionaries might also be constructed from lists or pairs `[`*key*`,` *value*`]` via the following command:

```
dict(['a0', 'b0'], ['a1', 'b1'], ...)
```

The two following commands, applied to the keys or to the dictionary itself are, by construction, equivalent to `D.values()`:

```
map (lambda t:D[t], D)    map (lambda t:D[t], D.keys())
```

The following test on the number of distinct values determines if the function represented by `D` is injective, `len(D)` being the number of dictionary entries:

```
sage: def injective(D):
....:   return len(D) == len (Set(D.values()))
```

The first two commands below build the target set $f(F)$ and the input set $f^{-1}(G)$ of subsets F and G of a function defined by the dictionary D; the last one constructs the dictionary `DR` corresponding to the inverse function f^{-1} of f, assumed to be bijective:

```
sage: Set([D[t] for t in F])
sage: Set([t for t in D if D[t] in G])
sage: DR = dict((D[t], t) for t in D)
```