SageMath introduction

Application of discrete models

SageMath (in short, Sage) is a computer algebra system, covering different areas of mathematics (algebra, calculus, number theory etc.). It is based on the Python programming language, i.e. the syntax and most functions of Python are available in Sage. In addition to this, Sage is built on top of many other specialized mathematical software (NumPy, SymPy, Maxima, GAP, R etc.), providing a common interface to them, and extending them with lots of new functionality.

Sage is free software ("free" as in "freedom"), e.g. it doesn't restrict the user in using the software.[1]

More information about Sage can be found on its website:

https://www.sagemath.org/

1 Accessing SageMath

The two main ways of using Sage is either to download and install it on a computer, or to connect to a Sage server on the Internet.

Installing Sage is not trivial, because it is a very complex software package. However, after doing it once, it's easier to use, faster, more reliable (you don't need an internet connection), and you have complete control over your computations.

Using an online Sage server is easier for the first time, but it suffers from the same problems as any other online service: you need constant internet connection to use it, that particular server must be always available, and you don't have control over your computations (you can only hope that they don't modify or delete your files, or send your data to someone else you don't want to). (Remember: using an online service means just using someone else's computer.)

1.1 Installing Sage

Below is a collection of some of the easiest ways to install Sage on your computer. Sage is primarily built for Linux-like systems, so it is somewhat easier to install it there, but it is also possible to do so on Windows.

Of course, the first step is to check whether Sage is already installed, e.g. on Linux, try to run the command '`sage`' in a terminal (without quotation marks).

[1] For the advantages of free software, see e.g.: https://www.gnu.org/philosophy/free-sw.html

1.1.1 As root on Linux

The easiest way to install Sage is if you have administrative (*root*) priviliges on a Linux distribution that has Sage in its repositories (usually named `sagemath`). Then you can install it in the usual way for that distribution, e.g. by using one of the following commands in the command line:

- Ubuntu, Debian: `sudo apt install sagemath`
- Arch Linux: `sudo pacman -S sagemath`
- Fedora: `sudo dnf install sagemath`

1.1.2 As a user on Windows/Linux

Otherwise, your best hope is to find a binary distribution of Sage. Unfortunately, the newest versions don't provide one anymore, but slightly older versions do, for both Windows and Linux (and others). (These versions should be perfectly fine for our purposes.) They can be found on the following page:

https://www.sagemath.org/mirrors.html

Choose any working mirror on this page (a *mirror* is a copy of a website to reduce load on the original server, thus improving download speed). From the *Archived (Outdated)* section, select the operating system (e.g. Windows or Linux), and download the appropriate (e.g. newest) file.

For Windows:

- Download the installer (which is an `.exe` file).
- Run it, and follow the instructions. (Note: you don't need to install the documentation, only the core package. Note 2: it asks you to accept the license, which is nonsense, as Sage is free software, so it doesn't restrict you using the software whether you accept the license or not.)
- Run the new SageMath icon on the desktop (if there are multiple icons, the one with the simplest name runs the Sage command line, i.e. which doesn't have "Shell" or "Notebook").

For Linux:

- Select your CPU architecture (most likely *64bit*).
- Download the appropriate compressed archive. (Officially they are for Debian and Ubuntu, but they may also work on other distributions, possibly after some fiddling. The distribution can be checked by the command '`cat /etc/*-release`'.)
- Extract the archive, e.g. by the command '`tar -xf <filename>.tar.bz2`' (to a place with enough space – the uncompressed package is around 12GB – and sufficient permissions).
- Run the executable file called `sage` from the extracted `SageMath` directory in the terminal. For example: '`cd <directory>/SageMath`', and then '`./sage`'.

On the first run, it may take a while for Sage to start and be usable. (And if you installed an older version, it might print some error messages like `DeprecationWarning` on the first run, but they can be safely ignored.)

1.1.3 Other ways

See the official documentation for other ways of installing Sage:
https://doc.sagemath.org/html/en/installation/index.html

1.2 Online Sage

1.2.1 SageMathCell

A very simple Sage server can be found at the following URL:

https://sagecell.sagemath.org/

Just type or copy the Sage code you want to run, and press Shift+Enter (or the Evaluate button) to see the result. The only way to save the code for later is to copy it to a text editor on your computer, and save it from there (the usual file extension is `*.sage`).

1.2.2 CoCalc

CoCalc provides a feature-rich web-based interface to SageMath (and other software):

https://cocalc.com/

You need to sign up with an account to use it. Also, it has several restrictions on your account unless you pay.

2 SageMath basics

The syntax of Sage is very similar to Python, but not 100% identical. The basics of Python will be discussed in the second half of this document. This part focuses on the command-line usage and mathematical capabilities of Sage.

For more information, see the official documentation of SageMath: https://doc.sagemath.org/

2.1 Sage command line

The Sage command line expects the commands one by one, and gives the result immediately. It can be used as a simple calculator, for example:

```
sage: 1+1
2
sage: 2^10
1024
sage: factorial(50)
30414093201713378043612608166064768844377641568960512000000000000
```

Here '`sage:`' is the *prompt*, i.e. the command can be written after it, and executed by pressing Enter.

We can already see that Sage has no problems with big numbers, e.g. it calculated all 65 digits of the above factorial (50!).

The caret (`^`) in Sage means exponentiation. (Note: this is different from Python, where it is denoted by two asterisks (`**`), e.g. in Python, `2**3 == 8`, while in Sage, `2^3 == 8`.)

We can create variables too:

```
sage: a = 2+2*4
sage: a
10
sage: (a^10 - 1) / (a - 1)
1111111111
```

Some useful controls in the Sage command line:

- To exit Sage, type '`quit`' or '`exit`', or press Ctrl+D.
- The "up" arrow key retrieves the last command (more "up" keys go further backwards, and "down" goes forwards). The retrieved command can be modified, and then rerun by pressing Enter. Pressing "up" after typing some text will retrieve the last command that started with this text.
- Question mark (`?`) gives information about a Sage function, e.g. '`factorial?`' (+ Enter) shows the documentation of the `factorial()` function. You can scroll the help by the arrow keys or Page Up/Down, and exit by Q.
- The Tab key completes the partially entered command, or shows the list of options. For example, after typing '`fa`', the Tab key will show a list of functions, including `factorial()`. Or after the assignment '`a=22`', typing '`a.`' in the next line and pressing Tab will show the list of member functions of '`a`' (e.g. '`a.binary()`' converts the number to binary).

- Underscore (_) refers to the previous result, double underscore (__) refers to the result before that, up to three underscores (___). For example:

```
sage: 2^3
8
sage: _ + 2
10
sage: (_ + 1) * __
88
```

2.2 Loading a .sage file

For a more complex program, it is better to write it to a separate file. Sage programs have the file extension `.sage`. (Any text editor can be used to create the Sage file. Most editors don't support Sage syntax highlighting, but they can be set to treat the file as Python.)

For example, create the file `new.sage` with the following content:

```
# Loop for these four numbers:
for n in [1,2,5,10]:
    print(n, 2^n, factorial(n))
```

A hash sign (#) introduces a *comment*, i.e. Sage ignores the rest of the line. Data can be printed by the `print()` function (because unlike the command line, here Sage doesn't print everything automatically). Loops and lists will be discussed in the second half of the document.

The Sage program can be loaded by the `load()` function:

```
sage: load("directory/new.sage")
1 2 1
2 4 2
5 32 120
10 1024 3628800
```

The path in `load()` is relative to the current directory, i.e. from which Sage was started. If the file is in this directory, then we can simply write '`load("new.sage")`'. If we don't know the current directory, we can query or change it using the following Sage commands (which are similar to the corresponding Linux commands):

- '`pwd`': print the current working directory;
- '`cd "/new/directory"`': change the current directly;
- '`ls`': list the files in the current directory.

But on Windows, directories are a bit more complicated: since Sage was built for Linux-like systems, it doesn't see Windows file systems directly. If Sage was installed on Windows as described above, then the drives `c:`, `d:` etc. are accessible under the directories `/cygdrive/c`, `/cygdrive/d` etc., and backslashes (\) must be changed to forward slashes (/). So for example, the file `d:\path\new.sage` can be accessed in Sage as `/cygdrive/d/path/new.sage`. Or if Sage was installed within WSL (Windows Subsystem for Linux), then `c:`, `d:` etc. are under `/mnt/c`, `/mnt/d` etc.

The `load()` function doesn't forget the previous state of the program (variables, functions etc.), but it runs the program on top of that state. This is usually not a problem, because the program – hopefully – defines all variables and functions it needs, so they will override their previous state anyway. But if somehow this still turns out to be a problem (e.g. because a variable has accidentally overridden a built-in function like `var()` or `sum()`), then the '`reset()`' function can be used to reset the state before the next '`load()`'.

2.3 Symbolic calculation

Sage calculates *symbolically* by default. What does it mean? For example:

```
sage: 2/3
2/3
```

This doesn't seem too useful yet. But try some more:

```
sage: 666/999
2/3
sage: 6/3
2
sage: 2/3 + 1/2
7/6
```

So it calculated everything, and gave the result in the simplest form.

But why didn't it give for `2/3` what most common calculators would give (and also Python): `0.6666666666666666`? Because this is not the exact value of 2/3. The exact value is 2/3, which can't be expressed any simpler than that. The exact value's decimal representation contains infinitely many 6's, which cannot be displayed by any calculator. Therefore, most common calculators use approximations. This is known as *numerical* calculation, as opposed to symbolic (or *exact*) calculation, which Sage does by default. The drawback of numerical calculation is that the rounding errors may add up, and the final result may be far from the exact value.

More examples for symbolic calculation in Sage (`sqrt(x)` $= \sqrt{x}$):

```
sage: sqrt(2)
sqrt(2)
sage: sqrt(16)
4
sage: sqrt(8)
2*sqrt(2)
sage: sqrt(2)*sqrt(8)
4
sage: sin(pi/3)
1/2*sqrt(3)
sage: log(e^3)
3
```

But Sage *can* calculate numerically if we ask it to do so, and for arbitrary precision:

```
sage: N(sqrt(2))
1.41421356237310
sage: N(sqrt(2), digits=80)
1.4142135623730950488016887242096980785696718753769480731766797379907324784621070
```

The result will also be numerical if the input is already a floating-point value:

```
sage: sqrt(2.)    # the decimal point (.) makes it numerical
1.41421356237310
```

But numerical calculation only gives approximate results:

```
sage: a = sqrt(2)
sage: b = N(sqrt(2))
sage: a^2 - 2
0                       # exact value
sage: b^2 - 2
4.44089209850063e-16    # approximation
```

But symbolic computation means more than just exact calculation with numbers. Sage can also calculate symbolically with letters, for example:

```
sage: (x+1)^2
(x + 1)^2
```

Again, it didn't do anything, but now because we didn't tell it what to do:

```
sage: expand((x+1)^2)
x^2 + 2*x + 1
sage: factor(x^4 - 1)
(x^2 + 1)*(x + 1)*(x - 1)
```

In Sage, `x` is a predefined *symbolic variable*. Other symbolic variables can be defined using the `var()` function:

```
sage: var("a,b")
(a, b)
sage: expand((a+b)^4)
a^4 + 4*a^3*b + 6*a^2*b^2 + 4*a*b^3 + b^4
```

Important: don't confuse symbolic variables with ordinary (Python-like) variables, because a variable always has an actual value (e.g. `5`), while a symbolic variable is the symbol itself (e.g. `x`):

```
sage: n = 5       # n is a variable
sage: n^2
25
sage: var("n");   # n is a symbolic variable
sage: n^2
n^2
```

And don't confuse symbolic variables with *symbolic constants* either: `pi`, `e`, `i`, `golden_ratio` etc., which have specific mathematical values, even though they can also be manipulated symbolically:

```
sage: x/4
1/4*x
sage: pi/4
1/4*pi       # so far they look the same
sage: tan(x/4)
tan(1/4*x)
sage: tan(pi/4)
1            # but not anymore
sage: N(e)   # symbolic constants have a numerical value
2.71828182845905
sage: N(x)   # symbolic variables don't
[...]
TypeError: cannot evaluate symbolic expression numerically
sage: i^2
-1
sage: e^(i*pi) + 1   # Euler's identity
0
```

2.4 Maths in Sage

Sage provides the following common mathematical functions:

`abs(x)`	$\lvert x\rvert$
`conjugate(x)`	$\overline{x}$
`factorial(n)`	$n!$
`binomial(n,k)`	$\binom{n}{k}$
`sqrt(x)`	$\sqrt{x}$
`sin(x)`	$\sin x$
`cos(x)`	$\cos x$
`tan(x)`	$\tan x$
`log(x,a)`	$\log_a x$
`log(x)`	$\ln x = \log_e x$
`exp(x)`	e^x
`floor(x)`	$\lfloor x \rfloor$
`ceil(x)`	$\lceil x \rceil$

```
sage: sqrt(-4)
2*I
sage: conjugate(2 + 3*i)
-3*I + 2
sage: binomial(5, 2)
10
sage: var("n"); binomial(n, 2)
1/2*(n - 1)*n
sage: exp(2*log(x))
x^2
sage: floor(57/10)
5
sage: (cos(x)^2 + sin(x)^2).simplify_full()
1
```

Note: in Sage, most mathematical functions can be written both as a regular function, e.g. `sqrt(4)`, and as a member function, e.g. `4.sqrt()`, `((x + 1)^2).expand()` etc.

Sage can do summation using the `sum()` function. It has two versions. The simpler one (inherited from Python) expects a list (or any other sequence):

```
sage: sum([10,20,30])
60
```

The other version works like the mathematical summation ($\sum$), and it expects four arguments: `sum(<formula>, <variable>, <from>, <to>)`. There is also a similar function for pruduct ($\prod$): `product(...)`. For example:

```
sage: var("n,k");
sage: sum(k^2, k, 1, 10)
385
sage: sum(k, k, 1, n).factor()
1/2*(n + 1)*n
sage: # oo == Infinity:
sage: sum(1/2^n, n, 0, oo)
2
sage: # product works similarly:
sage: product(k^2, k, 1, n)
factorial(n)^2
```

$$\sum_{k=1}^{10} k^2 = 1^2 + 2^2 + \ldots + 10^2 = 385$$

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

$$\sum_{n=0}^{\infty} \frac{1}{2^n} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots = 2$$

$$\prod_{k=1}^{n} k^2 = 1^2 \cdot 2^2 \cdot 3^2 \cdot \ldots \cdot n^2 = (n!)^2$$

Equations can be solved symbolically using the `solve()` function:

```
sage: solve(x^3 + 3*x^2 == 2*x + 2, x)
[x == -sqrt(2) - 2, x == sqrt(2) - 2, x == 1]
sage: _[0].rhs()   # extract the first solution from above
-sqrt(2) - 2
sage: var("x,y");
sage: solve([x + y == 6, x - y == 4], x, y)
[[x == 5, y == 1]]
sage: solve(2*x + y^2 == 6, x)
[x == -1/2*y^2 + 3]
sage: solve(2*x + y^2 == 6, y)
[y == -sqrt(-2*x + 6), y == sqrt(-2*x + 6)]
sage: # Infinitely many solutions will be given paremetrically:
sage: solve(x^2 + y^2 == 5, x, y)
[[x == r1, y == sqrt(-r1^2 + 5)], [x == r2, y == -sqrt(-r2^2 + 5)]]
sage: # Or we can restrict the solutions to integers:
sage: assume(x, y, "integer")    # also possible: "real", assume(x > 0) etc.
sage: solve(x^2 + y^2 == 5, x, y)
[(2, -1), (1, 2), (-1, -2), (2, 1), (-2, -1), (-2, 1), (1, -2), (-1, 2)]
```

Sage is also strong in calculus (limits, differentiation and integration):

```
sage: var("n");
sage: limit(2*n/(3*n + 1), n = oo)
2/3
sage: limit(1/(x-1), x = 1, dir='-')
-Infinity
sage: diff(x^3 + sin(2*x), x)
3*x^2 + 2*cos(2*x)
sage: diff(x^3 + sin(2*x), x, 2)
6*x - 4*sin(2*x)
sage: integral(3*x^2 + 2*cos(2*x), x)
x^3 + sin(2*x)
sage: integral(sin(x), x, 0, pi)
2
```

$$\lim_{n\to\infty} \frac{2n}{3n+1} = \frac{2}{3}$$

$$\lim_{x\to 1^-} \frac{1}{x-1} = -\infty$$

$$\big(x^3 + \sin(2x)\big)' = 3x^2 + 2\cos(2x)$$

$$\big(x^3 + \sin(2x)\big)'' = 6x - 4\cos(2x)$$

$$\int \big(3x^2 + 2\cos(2x)\big)\, dx = x^3 + \sin(2x) + C$$

$$\int_0^\pi \sin x\, dx = 2$$

Vectors can be created using the `vector()` function:

```
sage: v = vector([1,3,-2])
sage: v
(1, 3, -2)
sage: v[2]        # third component (counts from 0)
-2
sage: w = vector([2,1,0])
sage: v + 2*w
(5, 5, -2)
sage: v*w         # dot product
5
sage: v.norm(2)   # length (Euclidean norm)
sqrt(14)
```

And `matrix()` creates a matrix:

```
sage: M = matrix([[3,1,-4], [2,5,6], [1,4,8]])
sage: M
[ 3  1 -4]
[ 2  5  6]
[ 1  4  8]
sage: M[1][2]   # second row, third element
6
sage: M.transpose()
[ 3  2  1]
[ 1  5  4]
[-4  6  8]
sage: M.det()   # determinant
26
```

```
sage: M.inverse()
[  8/13 -12/13      1]
[ -5/13  14/13     -1]
[  3/26 -11/26    1/2]
sage: M.inverse() * M == matrix.identity(3)
True
sage: matrix([[1,-1,-1], [1,1,0], [3,0,1]]).eigenvalues()
[1, 1 - 2*I, 1 + 2*I]
```

And all of this works symbolically too, for example we can manipulate 2×2 matrices in general, using symbolic variables for the elements:

```
sage: var("a,b,c,d");
sage: M = matrix([[a,b], [c,d]])
sage: M
[a b]
[c d]
sage: M.det()
-b*c + a*d
sage: M^2
[a^2 + b*c a*b + b*d]
[a*c + c*d b*c + d^2]
sage: (M.inverse() * M).simplify_full()
[1 0]
[0 1]
sage: var("x,y");
sage: v = vector([x,y])
sage: M * v
(a*x + b*y, c*x + d*y)
sage: v.norm(2)
sqrt(abs(x)^2 + abs(y)^2)
```

Systems of linear equations can be solved in multiple ways:

```
sage: A = matrix([[0,1,-3], [4,5,-2], [2,3,-1]])
sage: b = vector([-5,10,7])
sage: A.solve_right(b)  # A*x == b  ("right": x multiplies from the right)
(-1, 4, 3)
sage: A*_ == b
True
sage: A.inverse() * b   # less efficient
(-1, 4, 3)
sage: var("x,y,z");
sage: solve([y - 3*z == -5, 4*x + 5*y - 2*z == 10, 2*x + 3*y - z == 7], x,y,z)
[[x == -1, y == 4, z == 3]]
```

3 Python programming

This section introduces the programming basics of Sage, inherited from Python. More information can be found in Python's official documentation: https://docs.python.org/.

3.1 Printing

Again, the `print()` function can be used to print something:

```
print("Hello world!")
```

Notice that Python doesn't require a semicolon (`;`) at the end of each statement (though it doesn't forbid it either). But it can be used to separate multiple statements in the same line:

```
print(a); print(a*2)
```

A number (or another object) can be printed in multiple ways:

```
print("a =", a)
print("a = " + str(a))   # what happens without str()?
print("a = %d" % a)      # like printf() in C
print("a = %d, a^2 = %d" % (a, a^2))
print(f"a = {a}, a^2 = {a^2}")
```

3.2 Types

Python (and so Sage) is a dynamically typed language, i.e. the types of the variables are detected at run-time when values are assigned to them, and they can be changed later with new assignments.

Here is a list of the most important types. The numbers are tricky, because they are different in Python and Sage.

Python only:

- `int`: integer, e.g. `123`
- `float`: floating-point number, e.g. `3.14`

Sage only:

- `Integer`: integer, e.g. `123`
- `RealNumber`: floating-point number, e.g. `3.14`
- `Rational`: rational number, e.g. `2/3`
- `Expression`: symbolic expression, e.g. `(x + 1)^2` or `sqrt(2)`

Python and Sage:

- `str`: string, i.e. sequence of characters, e.g. `"apple"`, `"123"` or `""` (empty string). Strings can be concatenated using the plus sign (`+`), e.g. `s = "app"; print(s + "le")`.
- `list`: list of elements, e.g. `[3, 1, 4, 1, 5]` (see later).
- `bool`: logical (boolean) value, i.e. either `True` or `False`. The comparison relations give a `bool` result, e.g. `b = 3 > 2` is the same as `b = True` (see also the `if` statements later).
- `NoneType`: its only value is `None`, i.e. "nothing". It indicates that a value is missing, e.g. a function doesn't return anything, or one of its parameters is undefined.

The type of a value can be queried using the `type()` function:

```
a = "apple"
print(type(a))               # <class 'str'>
print(type([1,2]) == list)   # True
```

Values can be converted to different types by using the name of the type as a function:

```
a = 12             # type(a) == Integer (or int in Python)
print(a == "12")   # False (an Integer is never equal to a string)
a = str(a)         # type(a) == str
print(a == "12")   # True
```

3.3 Branching (`if`)

In Python, a branch looks like this:

```
if a > 0:
    print("positive")
elif a == 0:
    print("zero")
else:
    print("negative")
```

As we can see, the structure of the program is determined not by symbols (like `{` and `}` in C), but by *indentation*, i.e. by the spaces and tabulators in the beginning of the line. The head of the control structure is separated from the body by a colon (`:`).

The control structure ends when the indentation is no longer greater than at the beginning of the control structure. For example:

```
if a > 2:
    print("This is printed if 'a' is greater than 2.")
    print("This too.")
if a <= 5:
    print("This is printed if 'a' is at most 5, regardless of the previous 'if'.")
    if a >= 1:
        print("This is printed if 'a' is between 1 and 5,")
        print("because this 'if' is inside the previous 'if' statement.")
    print("The inner 'if' ended, but the previous 'if' is still active,")
    print("so this is printed if a <= 5.")
print("This is always printed.")
```

(Attention: don't mix spaces and tabs! They might look similar to us, but the computer treats them differently!)

A simple '`if`' statement can also be written in one line:

```
if a == 4: print("four")
```

The condition of '`if`' can be any logical expression (i.e. `bool` value). It can be constructed using the relational operators: `<`, `>`, `<=`, `>=`, `==`, `!=`. Attention: the equality comparison is two equal signs (`if a == 5`), while the assignment is one (`a = 2`), don't mix them.

Logical expressions can be combined using the logical operators '`and`', '`or`' and '`not`', so for example the following statements are all equivalent:

```
if a >= 2 and a <= 5:     print("between 2 and 5")
if a >= 2 and not a > 5:  print("between 2 and 5")
if not (a < 2 or a > 5):  print("between 2 and 5")
```

If a variable is already a `bool`, it doesn't need to be compared to `True` or `False`, it can be used directly inside an '`if`':

```
b = a > 10   # b is either True or False
if b: print("OK")          # short and concise
if b == True: print("OK")  # same but redundant
# This is also possible (though not very useful):
if True: print("This is always printed.")
if False: print("This is never printed.")
```

More than that, Python allows not only a `bool`, but any expression inside an '`if`', without any comparison. Then the condition means that the value is "not zero", "not empty" etc. For example:

```
v = 12  # or v = "abc", or v = [1,2]
if v:     print("true")
if not v: print("false")
v = 0   # or v = "", or v = []
if v:     print("false")
if not v: print("true")
```

So '`if v`' means – depending on `v`'s type – '`if v != 0`', '`if v != ""`', '`if len(v) != 0`' etc., while '`if not v`' is the opposite ('`if v == 0`' etc.).

3.4 Loops (`for`, `while`)

The following loop prints the numbers from 10 to 19:

```
for i in range(10, 20):
    print(i)
```

In Python, the ranges include the start, but not the end, so the above loop goes only up to 19, even though the endpoint was given as 20. (Note: this is consistent with the indexing of lists, where, as we will see, a list with 10 elements can be indexed from 0 to 9.)

The following '`for`' loop iterates through all elements of the given list, and prints their squares (i.e. 9, 1, 16, 1 etc.):

```
for i in [3,1,4,1,5,9]:
    print(i*i)
```

After the 'in' keyword of the 'for' loop, we can give any sequence, e.g. a list, a string (which is a sequence of characters) or a generator (see the sequences later). One such sequence is range(), which generates an arithmetic series of integers, for example:

```
for i in range(2, 10): print(i)      # 2 3 4 5 6 7 8 9
for i in range(10): print(i)         # 0 1 2 3 4 5 6 7 8 9
for i in range(0, 10, 2): print(i)   # 0 2 4 6 8
for i in range(10, 0, -1): print(i)  # 10 9 8 7 6 5 4 3 2 1
for i in range(10, 0): print(i)      # (prints nothing, because 10 >= 0)
```

Loops can be nested, e.g. the following program prints a times table from 1 to 10:

```
for i in range(1, 11):
    for j in range(1, 11):
        print("%2d" % (i*j), end=" ")   # end: no line break
    print()   # line break after a row
```

If the sequence contains composite elements, the loop variable can also be equally composite:

```
for (x,y) in [(1,2),(-2,3),(5,6)]:
    if x > 0:
        print(f"x={x}, y={y}")
```

The 'while' loop is a *pre-test loop*, i.e. it runs as long as the condition is true. Its syntax is similar to the 'if' statement. For example, the following two loops are equivalent:

```
# for loop from 0 to 9:
for i in range(10):
    print(i)
# while loop from 0 to 9:
i = 0
while i < 10:
    print(i)
    i += 1    # or: i = i + 1  (Python has no 'i++')
```

Loops can contain 'break' and 'continue' statements like in other languages: 'break' breaks out of the loop, and continues the execution after it; 'continue' breaks only the current iteration of the loop, and jumps to the next iteration (e.g. next 'i' value).

Note: Python has no *post-test loop*, such as C's do-while loop.

3.5 Functions

In Python, a function can be defined as follows:

```
def add(a, b):
    return a + b
```

Then it can be used like this:

```
print(add(2, 3))
```

This is not very useful yet, so let's see something more complex:

```
def digits(n, base=10):
    dig = []
    while n > 0:
        dig.append(n % base)   # remainder
        n = n // base          # quotient
    dig.reverse()
    return dig
print(digits(127))             # [1, 2, 7]
print(digits(127, 2))          # [1, 1, 1, 1, 1, 1, 1]
print(digits(n=127, base=4))   # [1, 3, 3, 3]
```

Note: Python's syntax doesn't allow a function to be completely empty (nor a control structure like '`if`'). To leave it empty still (e.g. temporarily, if we want to write it later), we can use the '`pass`' command, which does nothing:

```
def something(n):
    pass
something(2)    # no effect and no error
```

Similarly:

```
if 3 == 3:
    pass
else:
    print("impossible")
```

3.6 Lists

One of the most important compound types of Python is the `list`. The simplest way to create a list is to give its elements between square brackets, e.g.:

```
[3, 1, 4, 1, 5]
[]                          # empty list
[123, [1,2,3], "apple"]   # inhomogenous list (the elements have different types)
```

Lists can be converted from other sequences using the `list()` function:

```
print(list(range(5)))    # [0, 1, 2, 3, 4]
print(list("apple"))     # ["a", "p", "p", "l", "e"]
```

Lists can also be created by a formula using the `for-in` construction, similarly to the mathematical set-builder notation (e.g. $\{x^2 \mid x \in \{0, \ldots, 4\}\}$):

```
print([i*i for i in range(5)])  # square numbers from 0 to 16: [0, 1, 4, 9, 16]
print([i*i for i in range(10) if i%2 != 0])  # odd square numbers from 1 to 81
print([2*i for i in [3,1,4,1,5]])  # [6,2,8,2,10]
print([[i*j for i in range(1,6)] for j in range(1,6)])  # times table with lists
```

The list elements can be queried or changed using the square bracket notation (indexing from 0):

```
L = [3,1,4,1,5,9]
print(L[0])   # 3
print(L[1])   # 1
print(L[2])   # 4
print(L[4])   # 5
print(L[6])   # IndexError: list index out of range
print(L[-1])  # 9   (counting backwards)
print(L[-2])  # 5
print(L[-3])  # 1
L[2] = 6
print(L)      # [3,1,6,1,5,9]
L[10] = 10    # IndexError: list assignment index out of range
```

The number of elements in a list can be queried by the `len()` function:

```
print(len([3,1,4,1,5]))   # 5
print(len([]))            # 0
```

The following two loops are equivalent (they both print the elements of the list):

```
L = [3,1,4,1,5,9]
for elem in L:
    print(elem)
for i in range(len(L)):
    print(L[i])
```

Elements can be added to or removed from a list:

```
L = [3,1,4,1,5]
L.append(9)      # insert 9 at the end
print(L)         # [3,1,4,1,5,9]
L.pop()          # remove the last element (9)
print(L)         # [3,1,4,1,5]
L.pop(2)         # remove L[2] (== 4)
print(L)         # [3,1,1,5]
L.insert(1, 9)   # insert 9 as the second element (L[1])
print(L)         # [3,9,1,1,5]
```

Lists can be concatenated and multiplied:

```
L1 = [3,1]
L2 = [4,1]
print(L1 + L2)   # [3,1,4,1]
print(3 * L1)    # [3,1,3,1,3,1]
print([2] * 5)   # [2,2,2,2,2]
```

Lists can be *sliced*, i.e. a range of elements can be queried:

```
L = [3,1,4,1,5,9]
print(L[3:5])    # [1,5]   (from L[3] up to but not including L[5])
print(L[3:])     # [1,5,9] (from L[3] until the end)
print(L[:3])     # [3,1,4] (all elements until L[3], not including that)
print(L[:])      # [3,1,4,1,5,9] (all elements, i.e. the copy of L, see below)
print(L[3::2])   # [1,9] (every second element starting from L[3])
print(L[::-1])   # [9,5,1,4,1,3] (all elements backwards)
L[-4:-1] = [2,2] # replace 3 elements from L[-4] with two elements: [2,2]
print(L)         # [3,1,2,2,9]
```

In Python, if a list (or other compound object) is assigned to another variable, it is not copied, but the same list will be referenced by both variables. To actually copy a list, use the `L[:]` construct. For example:

```
L1 = [3,1,4]
L2 = L1     # not a copy, but L1 and L2 refer to the same list
L2[1] = 2   # both are changed!
print(L1)   # [3,2,4]
print(L2)   # [3,2,4]
L3 = L1[:]  # now it's a copy, i.e. there will be two lists
L3[0] = 1   # only L3 is changed
print(L1)   # [3,2,4]
print(L3)   # [1,2,4]
```

3.7 Other sequences

A `tuple` is very similar to a list, but it cannot be modified after creation (i.e. it is *immutable*). Otherwise it works in the same way, except that it is created using round brackets:

```
T = (3,1,4)
print(T[-1])      # 4
print(T[1:])      # (1,4)
print(T + (1,5))  # (3,1,4,1,5)
print(2*T)        # (3,1,4,3,1,4)
T[2] = 6          # TypeError: 'tuple' object does not support item assignment
```

(A tuple with two elements is also called a *pair*.)

A string (`str`) is a sequence of characters. It can be given between quotes, in multiple ways (the first two lines are equivalent):

```
print("I'm OK.")
print('I\'m OK.')
print("""multi-line
string can be given
like this""")
```

Strings have similar operations than lists, except that they are *immutable* like tuples. Strings also support some additional operations:

```
S = "text"
print(S[2])           # "x"
print(S[1:-1])        # "ex"
print("a"*10 + "bc")  # "aaaaaaaaaabc"
print(S.replace("t", "T"))         # "TexT" (but S itself didn't change)
print("abc,d,e,,fg".split(","))    # ["abc", "d", "e", "", "fg"]
print(",".join(["a","bc","def"]))  # "a,bc,def"
```

A *generator* is an abstract sequence which generates its elements one by one, but without storing them all at once in the memory (unlike e.g. a list). An example is `range()`, which is typically used in the '`for`' loops. For example, `range(1000000)` doesn't store all numbers from 0 to 999999 in the memory, only one at a time. (If we really want to, we can force Python to store all these numbers by converting the generator to a list: `list(range(1000000))`.)

A generator can be defined similarly to a function, but instead of using '`return`' (which ends the function immediately), the '`yield`' command is used, which gives the values one by one. For example, the following generator replicates `range(a,b)`:

```
def myrange(a, b):
    i = a
    while i < b:
        yield i
        i += 1
```

Then it can be used just like `range()`:

```
for i in myrange(2, 10):
    print(i*i)
```

Since a generator doesn't store all the elements, it can even be infinite:

```
def squares():
    i = 1
    while True:   # infinite loop
        yield i*i
        i += 1
```

Of course, a loop that uses an infinite generator must terminate itself:

```
for x in squares():
    if x > 1000: break
    print(x)
```

3.8 Testing

To make sure that a program is correct, it must be *tested*. For example, one might try to implement multiplication by using repeated addition, and may write it in the following – inefficient and incorrect – way:

```
def mul (a, b):
    c = a
    for i in range(1, b):
        c += a
    return c
```

If we try it once, it seems to work:

```
sage: mul(24, 100)
2400
```

But let's do something better! For example, let's test it for all numbers between 0 and 100:

```
for a in range(101):   # don't forget: range(n) ends at n-1
    for b in range(101):
        assert mul(a,b) == a*b
```

The '`assert`' statement checks whether the given condition is true, and if not, then it prints an error message and terminates the program. (The same thing could be done by using an '`if`', a `print()` and several '`break`' statements, but an '`assert`' is much simpler.)

Running the above program gives a long error message, ending in the following line:

```
AssertionError:
```

But it didn't tell which numbers it failed for. For that, we need to extend '`assert`':

```
for a in range(101):
    for b in range(101):
        assert mul(a,b) == a*b, f"mul({a},{b})"
```

The second, optional parameter of '`assert`' is a string, which is printed when the condition fails. (Here we gave a formatted string that contains the values of `a` and `b`, see the beginning of the Python part of this document.) Then the error message will end like this:

```
AssertionError: mul(1,0)
```

And indeed, `mul(1,0) == 1`, not `0` as it should be. The function doesn't work properly for `0`. It can be fixed by starting the summation from `0`, and then do one more iteration with the loop:

```
def mul (a, b):
    c = 0
    for i in range(b):
        c += a
    return c
```

Now all tests pass.

But the above testing method has its limitations, because it is too slow for bigger numbers. Instead (or additionally), we can test the function randomly. Random numbers can be generated by the `randrange()` function, which expects the same parameters as `range()`:

```
sage: randrange(100)            # random number between 0 and 99
94
sage: randrange(100, 200, 10)   # 100, 110, 120, ..., 190
180
```

(Note: in Python, `randrange()` needs the following line to be inserted at the beginning of the file: '`from random import randrange`' – but it is not necessary in Sage, where it is always available.)

For example, we can test our function using random numbers as follows:

```
for i in range(1000):
    a = randrange(10000)
    b = randrange(10000)
    assert mul(a,b) == a*b, f"mul({a},{b})"
```

After all this testing, we can be confident enough that our function works correctly. But there is another important aspect: how fast is it? In Sage, we can measure the running time of a statement using the '`time`' command, for example:

```
sage: time mul(1000, 123456789)
CPU times: user 12.9 s, sys: 0 ns, total: 12.9 s
Wall time: 12.9 s
123456789000
sage: time 1000 * 123456789
CPU times: user 16 µs, sys: 0 ns, total: 16 µs
Wall time: 21.5 µs
123456789000
```

It printed a lot of information, but let's just pay attention to the "Wall time" (i.e. the actual elapsed time): the `mul()` function took ∼13 seconds, while the multiplication operator took ∼21 *microseconds*, i.e. it was basically instantaneous (ms = 1/1000 s, µs = 1/1 000 000 s, ns = 1/1000 000 000 s). So our function is – unsurprisingly – very slow.

4 Author

This document was written by Uray M. János, partly using the following sources:

[1] https://compalg.elte.gitlab-pages.hu/dimoa-web/tematika/dimoa (Hungarian)

[2] https://doc.sagemath.org/

[3] https://docs.python.org/

[4] https://en.wikipedia.org/wiki/SageMath

Errors and suggestions can be reported here: uray.janos@inf.elte.hu.