Definition and concept of a list in Python

Concept of a list and definition *(not new)*

“A list is a (finite) ***ordered sequence*** *of* ***any*** *kind of Python* ***values****.”*

Goal ?

⟶ efficiently and easily **store** any kind of data, that can be **accessed** and **modified** later.

Practice by yourself (online)?

You should try to **practice a little bit by yourself**, *additionally* to the labs. That page IntroToPython.org/lists_tuples.html contains great exercises!

Definition and concept of a list in Python

Concept of a list and definition *(not new)*

> *"A list is a (finite)* ***ordered sequence*** *of* ***any*** *kind of Python* ***values****."*

Goal ?

⟶ efficiently and easily **store** any kind of data, that can be **accessed** and **modified** later.

Practice by yourself (online)?

You should try to **practice a little bit by yourself**, *additionally* to the labs. That page IntroToPython.org/lists_tuples.html contains great exercises!

Defining, accessing, modifying a list

Defining a list and reading its elements *(not new)*

- it is easy to **define a list**:
  - empty `l = []`,
  - or not `l = [3, 4, 5]`, `team = ['Rahul', 'Nikitha']` etc,
- and to **read** its elements: `l[0]`, . . . , `l[n-1]` (if `n = len(l)` is the **length** of the list, ie. its size)).

Two important **warnings** *(new!)*

- indexing **starts from** 0 **to** $n - 1$ and not from 1 !
- indexing errors can (and will) happen: for $k > n - 1$, `l[k]` raises a `IndexError: list index out of range` .

Modifying a list in place *(not new)*

- **modifying one element**: `l[0] = 6` makes `l` becoming `[6, 4, 5]`,
- modifying a **slice** (a sub-list) will be seen after.

Defining, accessing, modifying a list

Defining a list and reading its elements *(not new)*

- it is easy to **define a list**:
  - empty `l = []`,
  - or not `l = [3, 4, 5]`, `team = ['Rahul', 'Nikitha']` etc,
- and to **read** its elements: `l[0]`, . . . , `l[n-1]`
  (if `n = len(l)` is the **length** of the list, ie. its size)).

Two important **warnings** *(new!)*

- indexing **starts from** 0 **to** $n-1$ and not from 1 !
- indexing errors can (and will) happen: for $k > n-1$,
  `l[k]` raises a `IndexError: list index out of range` .

Modifying a list in place *(not new)*

- **modifying one element**: `l[0] = 6` makes `l` becoming `[6, 4, 5]`,
- modifying a **slice** (a sub-list) will be seen after.

One classic example of lists: arithmetical progressions

range creates (finite) arithmetical progressions *(not new)*

The range function can be used, in three different ways:

- range(n) $= [0, 1, .., n-1]$,
- range(a, b) $= [a, a+1, .., b-1]$,
- range(a, b, k) $= [a, a+k, a+2k, .., a+i \times k]$ (last i with $a + ik < b$).

Default values are $a = 0$ and $k = 1$, and $k \neq 0$ is required.

Useful for loops! *(not new)*

For example, to print the square of the first 30 odd integers:

```
for i in range(1, 31, 2):
    # 1 <= i < 1 + 2*(31/2) - 1
    print i, "**2 is", i**2
```

Remark: if we just use range in a **for** loop, the xrange function is better (more time and memory efficient).

One classic example of lists: arithmetical progressions

`range` creates (finite) arithmetical progressions *(not new)*

The `range` function can be used, in three different ways:

- `range(n)` $= [0, 1, .., n-1]$,
- `range(a, b)` $= [a, a+1, .., b-1]$,
- `range(a, b, k)` $= [a, a+k, a+2k, .., a+i \times k]$ (last i with $a + ik < b$).

Default values are $a = 0$ and $k = 1$, and $k \neq 0$ is required.

Useful for loops! *(not new)*

For example, to print the square of the first 30 odd integers:

```
for i in range(1, 31, 2):
    # 1 <= i < 1 + 2*(31/2) - 1
    print i, "**2 is", i**2
```

Remark: if we just use `range` in a **for** loop, the `xrange` function is better (more time and memory efficient).

Looping over a list

To loop over a list, there are 3 ways *(new!)*

- `for i in range(len(l)):` *(not new)*
  then the values of `l` can be obtained with `l[i]` (one by one),
- `for x in l:` *(not new)*
  then the values of `l` are just `x` (one by one),
- `for i, x in enumerate(l):` *(new!)*
  then the values of `l` are just `x` (one by one),
  but can be modified with `l[i] = newvalue`
- Warning: modifying this `x` will **not** change the list!

Example of a simple `for` loop

```
for name in ['Awk Girl', 'Batman', 'Wonder Woman']:
    print name, "is a member of the JLA."
```

Looping over a list

To loop over a list, there is 3 approaches *(new!)*

- `for i in range(len(l)):` *(not new)*
  then the values of `l` can be obtained with `l[i]` (one by one),
- `for x in l:` *(not new)*
  then the values of `l` are just `x` (one by one),
- `for i, x in enumerate(l):` *(new!)*
  then the values of `l` are just `x` (one by one),
  but can be modified with `l[i] = newvalue`
- Warning: modifying this `x` will **not** change the list!

Example of a simple `for` loop

```
for name in ['Awk Girl', 'Batman', 'Wonder Woman']:
    print name, "is a member of the JLA."
```

Negative indexing and slicing for a list

Negative indexing of a list `l`? *(new!)*

We can read its elements **from the end** with negative indexes.

- Instead of writing `l[n-1]`, write `l[-1]`: it is simpler!
- Similarly `l[-2]` is like `l[n-2]` etc.

Warning: indexing errors can still happen, `l[-k]` raises a `IndexError: list index out of range` when $k > n$.

Slicing for a list? `l` *(new!)*

Slicing a list is useful to **select a sub-list** of the list: `l[first:bound:step]`. By default, `first` is 0 (inclusive), `bound` is n (exclusive), `step` is 1.

- reading a slice: `l[0:3]`, `l[2:]` or `l[:3]` or `l[0::2]` for examples.
- modifying a slice: `l[:5] = [0]*5` for example puts a 0 in each of the 5 values `l[i]` for $0 \leqslant i < 5$,
- Warning: modifying a slice with a list of different size **might** raise an error like `ValueError: attempt to assign sequence of size 5 to extended slice of size 4` (it is a tricky point, be cautious).

Negative indexing and slicing for a list

Negative indexing of a list `l`? *(new!)*

We can read its elements **from the end** with negative indexes.

- Instead of writing `l[n-1]`, write `l[-1]`: it is simpler!
- Similarly `l[-2]` is like `l[n-2]` etc.

Warning: indexing errors can still happen, `l[-k]` raises a `IndexError: list index out of range` when $k > n$.

Slicing for a list? `l` *(new!)*

Slicing a list is useful to **select a sub-list** of the list: `l[first:bound:step]`. By default, `first` is 0 (inclusive), `bound` is n (exclusive), `step` is 1.

- reading a slice: `l[0:3]`, `l[2:]` or `l[:3]` or `l[0::2]` for examples.
- modifying a slice: `l[:5] = [0]*5` for example puts a 0 in each of the 5 values `l[i]` for $0 \leqslant i < 5$,
- Warning: modifying a slice with a list of different size **might** raise an error like `ValueError: attempt to assign sequence of size 5 to extended slice of size 4` (it is a tricky point, be cautious).

Some functions for lists

We give here some **functions for lists**, already seen and used in labs.

5 useful functions *(not new)*

- `len` gives the *length* of the list (and `len([]) = 0`),
- `min` and `max` returns the *minimum* and the *maximum* of the list. Might raise `ValueError: min() arg is an empty sequence`.
- `sum` computes the *sum* of the values in the list. Warning: there is **no** `prod` function to compute the product!
- `sorted` sorts the list (if possible, in $O(n \log(n))$ in the worst case), and returns **a new copy** of the list, sorted in the increasing order.

And more functions are available!

- `all` (resp. `any`) computes the **boolean** $\forall x \in mylist$ (resp. $\exists x \in mylist$),
- `filter` and `map` are not really used in practice,
- and `reduce` is . . . more complicated.

Some functions for lists

We give here some **functions for lists**, already seen and used in labs.

5 useful functions *(not new)*

- `len` gives the *length* of the list (and `len([]) = 0`),
- `min` and `max` returns the *minimum* and the *maximum* of the list.
  Might raise `ValueError: min() arg is an empty sequence`.
- `sum` computes the *sum* of the values in the list.
  Warning: there is **no `prod`** function to compute the product!
- `sorted` sorts the list (if possible, in $O(n \log(n))$ in the worst case), and returns **a new copy** of the list, sorted in the increasing order.

And more functions are also available!

-

Some methods for lists

2 convenient notations: *(new!)*

- `l1 + l2` is the concatenation of the two lists `l1` and `l2`,
- `l * k` is like `l + l + ... + l`, k times. Example: `l = [0] * 100`.

Some **methods** for lists can be useful.

7 simple methods: *(new!)*

- `l.sort()` sorts the list `l` **in place** (ie. modifies the list),
- `l.append(newvalue)` adds the value `newvalue` at the end
- `l.pop()` removes and returns the last item,
- `l.index(x)` returns the first index of value `x`. Will raise ValueError if the value `x` is not present in `l`,
- `l.count(y)` counts how many times the value `y` is present,
- `l.extend(otherlist)` is like `l = l + otherlist`,
- `l.insert(index, z)` will insert the new value `z` at position `index`.

Some methods for lists

2 convenient notations: *(new!)*

- `l1 + l2` is the concatenation of the two lists `l1` and `l2`,
- `l * k` is like `l + l + ... + l`, k times. Example: `l = [0] * 100`.

Some **methods** for lists can be useful.

7 simple methods: *(new!)*

- `l.sort()` sorts the list `l` **in place** (ie. modifies the list),
- `l.append(newvalue)` adds the value `newvalue` at the end
- `l.pop()` removes and returns the last item,
- `l.index(x)` returns the first index of value `x`.
  Will raise ValueError if the value `x` is not present in `l`,
- `l.count(y)` counts how many times the value `y` is present,
- `l.extend(otherlist)` is like `l = l + otherlist`,
- `l.insert(index, z)` will insert the new value `z` at position `index`.

Sum-up about **lists**, and what for tomorrow?

About lists, we saw: *(new!)*

- concept of a list in Python, and how to define it, modify it or its elements and read them,
- some new concepts, like **negative indexing** or **slicing**,
- looping, 3 approaches, and `enumerate(l)` is new,
- functions for lists (`len`, `max`/`min`, `sum`, `sorted` . . .),
- methods for lists (`sort`, `append`, `pop`, `index`, `count`, `extend` etc . . .).

What is next ?

Tomorrow: matrices as **list of line vectors**, some more list comprehensions, and one nice example will be seen (with a *list* of your *grades*).
We will then introduce **sets** in Python, like `s = { -1, 1 }`.

Sum-up about **lists**, and what for tomorrow?

About lists, we saw: *(new!)*

- concept of a list in Python, and how to define it, modify it or its elements and read them,
- some new concepts, like **negative indexing** or **slicing**,
- looping, 3 approaches, and `enumerate(l)` is new,
- functions for lists (`len`, `max`/`min`, `sum`, `sorted` ...),
- methods for lists (`sort`, `append`, `pop`, `index`, `count`, `extend` etc ...).

What is next ?

Tomorrow: matrices as **list of line vectors**, some more list comprehensions, and one nice example will be seen (with a *list* of your *grades*).
We will then introduce **sets** in Python, like `s = { -1, 1 }`.

Introduction to **list comprehensions**

Python offers a nice and efficient syntax for **easily defining a list of values** obtained with an expression of one index.

What is a **list comprehension**? *(new!)*

The syntax is like this: `[` "expression with i" `for i in somelist ]`

- List of the first 100 triangular numbers?
  ↪ `[ k*(k+1)/2 for k in range(100) ]`,
- Quickly compute a partial sum of a series? $S_{10000} = \sum_{n=1}^{10000} 1/n^2$
  ↪ `S = sum([ 1.0/(n**2) for n in range(1,10000+1) ])`,
- Sum of numbers below 10000 that are multiples of 11 **or** multiples of 7?
  ↪ `sum([ k for k in range(1,10000) if k%7==0 or k%11==0 ])`
- and many more examples are possible . . .

Introduction to **list comprehensions**

Python offers a nice and efficient syntax for **easily defining a list of values** obtained with an expression of one index.

What is a **list comprehension**? *(new!)*

The syntax is like this: `[` "expression with i" `for i in somelist ]`

- List of the first 100 triangular numbers?
  ↪ `[ k*(k+1)/2 for k in range(100) ]`,

- Quickly compute a partial sum of a series? $S_{10000} = \sum_{n=1}^{10000} 1/n^2$
  ↪ `S = sum([ 1.0/(n**2) for n in range(1,10000+1) ])`,

- Sum of numbers below 10000 that are multiples of 11 **or** multiples of 7?
  ↪ `sum([ k for k in range(1,10000) if k%7==0 or k%11==0 ])`

- and many more examples are possible ...

Introduction to **list comprehensions**

Python offers a nice and efficient syntax for **easily defining a list of values** obtained with an expression of one index.

What is a **list comprehension**? *(new!)*

The syntax is like this: `[` "expression with i" `for i in somelist ]`

- List of the first 100 triangular numbers?
  ↪ `[ k*(k+1)/2 for k in range(100) ]`,
- Quickly compute a partial sum of a series? $S_{10000} = \sum_{n=1}^{10000} 1/n^2$
  ↪ `S = sum([ 1.0/(n**2) for n in range(1,10000+1) ])`,
- Sum of numbers below 10000 that are multiples of 11 **or** multiples of 7?
  ↪ `sum([ k for k in range(1,10000) if k%7==0 or k%11==0 ])`
- and many more examples are possible …

Matrices as list of lists

Matrix as list of line vectors *(new!)*

$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ has three lines, $x_1 = [1\ 2\ 3]$, $x_2 = [4\ 5\ 6]$, $x_3 = [7\ 8\ 9]$.

So in Python, this matrix can be written as `M = [x_1, x_2, x_3]`, with:
`M[0]=x_1=[1, 2, 3]`, `M[1]=x_2=[4, 5, 6]`, `M[2]=x_3=[7, 8, 9]`.
`M = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]` is a list of 3 lists (of sizes 3).

Examples of list comprehension for matrices *(new!)*

- trace(M) is `sum([ M[i][i] for i in range(len(M)) ])`,
- $A + B$ is `[ [ A[i][j] + B[i][j] for i in range(len(A)) ] for j in range(len(A)) ]` (if A and B are square),
- ... and you will figure out $A \times B$ by yourself (in this week lab).

Matrices as list of lists

Matrix as list of line vectors *(new!)*

$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ has three lines, $x_1 = [1\ 2\ 3]$, $x_2 = [4\ 5\ 6]$, $x_3 = [7\ 8\ 9]$.

So in Python, this matrix can be written as `M = [x_1, x_2, x_3]`, with:
`M[0]=x_1=[1, 2, 3]`, `M[1]=x_2=[4, 5, 6]`, `M[2]=x_3=[7, 8, 9]`.
`M = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]` is a list of 3 lists (of sizes 3).

Examples of list comprehension for matrices *(new!)*

– trace(M) is `sum([ M[i][i] for i in range(len(M)) ])`,

– $A + B$ is `[ [ A[i][j] + B[i][j] for i in range(len(A)) ] for j in range(len(A)) ]` (if A and B are square),

– … and you will figure out $A \times B$ by yourself (in this week lab).

Matrices as list of lists

Matrix as list of line vectors *(new!)*

$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ has three lines, $x_1 = [1\ 2\ 3]$, $x_2 = [4\ 5\ 6]$, $x_3 = [7\ 8\ 9]$.

So in Python, this matrix can be written as `M = [x_1, x_2, x_3]`, with:
`M[0]=x_1=[1, 2, 3]`, `M[1]=x_2=[4, 5, 6]`, `M[2]=x_3=[7, 8, 9]`.
`M = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]` is a list of 3 lists (of sizes 3).

Examples of list comprehension for matrices *(new!)*

- trace(M) is `sum([ M[i][i] for i in range(len(M)) ])`,
- $A + B$ is `[ [ A[i][j] + B[i][j] for i in range(len(A)) ] for j in range(len(A)) ]` (if A and B are square),
- … and you will figure out $A \times B$ by yourself (in this week lab).

Other data structures similar to list : tuples

About tuples

Tuples are exactly like lists,
but **cannot be modified after being created**:

- A **tuple** is an **unmutable list**.
- A tuple is written `s = ()` for the *empty* tuple, or `t = (x, y)` For example, `v = (1, 0, 1)` is like a vector of $\mathbb{R}^3$.
- **Type conversion** between tuples and lists can be done: with `t = tuple(mylist)` and `l = list(mytuple)` . . . !

Other data structures similar to list : strings

About strings

A **string** is *almost* like a **list of characters**:
`name = 'batman'` is like `['b', 'a', 't', 'm', 'a', 'n']`:

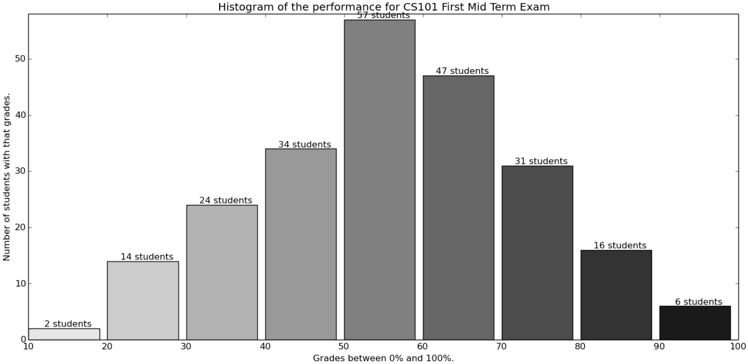
- Accessing and slicing is done the same way for **strings**: `name[0]` is `'b'`, `name[3:]` is `'man'` etc,
- Looping over a string will loop **letter by letter** (`'b'`,`'a'` etc).
- But warning: a string is **unmutable**! `name[0] = 'C'` fails, `name = 'C' + name[1:]` is good

One example of use of a list (cf. Spyder demo)

Using a list to store values, e.g. grades of the first Mid Term Exam

The demo plots an histogram for your grades, written as a Python list:

```
# That list has 230 values between 0 and 100
grades = [ 36, 73.5, ..., 34, 56, 68, 61, 29 ]
```


Histogram of the performance for CS101 First Mid Term Exam
Number of students with that grades.
Grades between 0% and 100%.
2 students
14 students
24 students
34 students
57 students
47 students
31 students
16 students
6 students