

Séquences et boucles*

Tristan Crolard

Laboratoire CEDRIC
Equipe « Systèmes Sûrs »

tristan.crolard@cnam.fr

cedric.cnam.fr/sys/crolard

*. Ces supports sont adaptés de *Python for Computational Science* (2024)

Sequences – overview

Different types of sequences:

- ▶ strings (immutable)
- ▶ lists (mutable)
- ▶ tuples (immutable)

They share some **common primitives**.

Strings

```
>>> a = "Hello World"
```

```
>>> type(a)
```

```
<class 'str'>
```

```
>>> len(a)
```

```
11
```

```
>>> print(a)
```

```
Hello World
```

Different possibilities to delimit strings:

```
'A_string'          # not recommended, except a single character  
"Another_string"   # double quotes are perfect for messages  
"A_string_with_a'_in_the_middle" # with single quotes  
"""A_string_with_triple_quotes  
can extend over several  
lines"""
```

Strings – exercise

- ▶ Define a, b and c at the Python prompt:

```
>>> a = "One"
```

```
>>> b = "Two"
```

```
>>> c = "Three"
```

- ▶ Exercise: what do the following expressions evaluate to?

- `a + b + c`

- `5 * b` # python idiom

- `a[0], a[1], a[2]` # indexing

- `b[-1]` # python idiom

- `c[1:2]` # slicing

Lists

```
>>> [] # the empty list
```

```
[]
```

```
>>> [42] # a 1-element list
```

```
[42]
```

```
>>> ['A', 'B', 'C'] # a 3-element list
```

```
['A', 'B', 'C']
```

```
>>> [[1, 2], [3, 4, 5], [6]] # a list of lists
```

```
[[1, 2], [3, 4, 5], [6]]
```

```
>>>
```

- ▶ Lists are **sequences** of items
- ▶ Access through index, and slicing (as for strings)
- ▶ Lists are **mutable**

Example: accessing and mutating lists

```
>>> a = [] # creates a list
```

```
>>> a += ["dog"] # augmented assignment
```

```
>>> a += ["cat", "mouse"]
```

```
>>> a
```

```
['dog', 'cat', 'mouse']
```

```
>>> a[2] = "horse" # re-assign a[2]
```

```
>>> a
```

```
['dog', 'cat', 'horse']
```

```
>>> print(a[0]) # access first element (with index 0)
```

```
dog
```

```
>>> print(a[-1])
```

```
horse
```

Example: mutating lists using slices

```
>>> a
```

```
['dog', 'cat', 'horse']
```

```
>>> a += ["mouse", "snake"]           # augmented assignment
```

```
>>> a
```

```
['dog', 'cat', 'horse', 'mouse', 'snake']
```

```
>>> a[3:3] = ["fish", "bird"]       # insertion
```

```
>>> a
```

```
['dog', 'cat', 'horse', 'fish', 'bird', 'mouse', 'snake']
```

```
>>> a[2:6] = []                       # deletion
```

```
>>> a
```

```
['dog', 'cat', 'snake']
```

Packing and unpacking lists

```
>>> l1 = ['dog', 'cat', 'horse']
```

```
>>> l2 = [*l1, 'snake']
```

```
>>> l2
```

```
['dog', 'cat', 'horse', 'snake']
```

```
>>> l3 = ['fish', 'bird']
```

```
>>> l4 = [*l2, *l3]
```

```
>>> l4
```

```
['dog', 'cat', 'horse', 'snake', 'fish', 'bird']
```

```
>>> l5 = [*l2, 'tiger', *l3]
```

```
>>> l5
```

```
['dog', 'cat', 'horse', 'snake', 'tiger', 'fish', 'bird']
```

```
>>> l5
```

```
['dog', 'cat', 'horse', 'snake', 'tiger', 'fish', 'bird']
```

```
>>> [head, *tail] = l5
```

```
>>> head
```

```
dog
```

```
>>> tail
```

```
['cat', 'horse', 'snake', 'tiger', 'fish', 'bird']
```

```
>>> [*prefix, last] = l5
```

```
>>> prefix
```

```
['dog', 'cat', 'horse', 'snake', 'tiger', 'fish']
```

```
>>> last
```

```
bird
```

List methods

Since Python is also an object-oriented language, primitives are often defined as **methods**. In particular, most primitives on lists (and strings) are methods.

The syntax of a method call is:

`o.f(x1, . . . , xn)`

where object `o` is a distinguished argument of method `f`.

Note. We will not define any method in this course, but you still need to learn how to use some primitive methods from the standard library.

Example: mutating lists using methods

```
>>> a = ["dog", "cat"]
```

```
>>> a.append("horse")
```

```
>>> a
```

```
['dog', 'cat', 'horse']
```

```
>>> a.extend(["mouse", "snake"])
```

```
>>> a
```

```
['dog', 'cat', 'horse', 'mouse', 'snake']
```

```
>>> print(a.pop())
```

```
snake
```

```
>>> a
```

```
['dog', 'cat', 'horse', 'mouse']
```

```
>>> print(a.pop(2))
```

```
horse
```

```
>>> a
```

```
['dog', 'cat', 'mouse']
```

Example: lists containing lists

```
>>> a = [[1], [1, 2], [1, 5, 10], [1, 10, 100, 1000]]
```

```
>>> a
```

```
[[1], [1, 2], [1, 5, 10], [1, 10, 100, 1000]]
```

```
>>> a[3]
```

```
[1, 10, 100, 1000]
```

```
>>> max(a[3])
```

```
1000
```

```
>>> min(a[3])
```

```
1
```

```
>>> a[3][2]
```

```
100
```

Sequences – str examples

```
>>> a = "hello_world"
```

```
>>> print(a[4])
```

o

```
>>> print(a[4:7])
```

o w

```
>>> len(a)
```

11

```
>>> 'd' in a
```

True

```
>>> print(a + "!!!")
```

hello world hello world

Strings are immutable

- ▶ Strings are immutable:

```
>>> a = "hello_world" # String example
```

```
>>> a[3] = 'x'
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

- ▶ You need to create a new string instead, for instance:

```
>>> a = a[0:3] + 'x' + a[4:]
```

```
>>> a
```

```
helxo world
```

Sequences – summary

- ▶ lists and strings are sequences.
- ▶ sequences share the following operations:

<code>a[i]</code>	returns element with index <code>i</code> of <code>a</code>
<code>a[i:j]</code>	returns elements <code>i</code> up to <code>j - 1</code>
<code>len(a)</code>	returns number of elements in sequence <code>a</code>
<code>min(a)</code>	returns smallest value in sequence <code>a</code>
<code>max(a)</code>	returns largest value in sequence <code>a</code>
<code>x in a</code>	returns <code>True</code> if <code>x</code> is an element in <code>a</code>
<code>a + b</code>	concatenates <code>a</code> and <code>b</code>
<code>n * a</code>	creates <code>n</code> copies of sequence <code>a</code>

In this table, `a` and `b` are sequences, `i`, `j` and `n` are integers, `x` is an element.

Tuples

- ▶ tuples are **immutable** (unchangeable) whereas lists are **mutable**
- ▶ tuples are usually written using parentheses (“round brackets”):

```
>>> t = (3, True, "bob")          # t for Tuple
```

```
>>> t
```

```
(3, True, 'bob')
```

```
>>> type(t)
```

```
<class 'tuple'>
```

Tuples are defined by the comma

- ▶ tuples are defined by the comma (!), not the parentheses

```
>>> t = 3, True, "bob"
```

```
>>> t
```

```
(3, True, 'bob')
```

```
>>> type(t)
```

```
<class 'tuple'>
```

- ▶ the parentheses are optional (but they should be written anyway)

When do we use tuples?

1. use tuples if you want to make sure that a set of values doesn't change.
2. using tuples, we can assign several variables in one line (known as **tuple unpacking**)

```
>>> (x, y, z) = (0, 0, 1)
```

In particular, this allows for “instantaneous swap” of values:

```
>>> (x, y) = (y, x)
```

3. functions can return tuples when they need to return several objects:

```
>>> def f(x: int) -> tuple[int, float]:  
    return (x*2, x/2)
```

```
>>> f(10)
```

```
(20, 5.0)
```

Loops – introduction

- ▶ Computers are good at repeating tasks (often the same task for many different sets of data).
- ▶ Loops are the way to execute the same (or very similar) tasks repeatedly (“in a loop”).
- ▶ Python provides the “for loop” and the “while loop”.

Example program: for-loop

```
>>> animals = ["dog", "cat", "mouse"]
      for animal in animals:
          print("This is the " + animal + "!")
```

This is the dog!

This is the cat!

This is the mouse!

The for-loop *iterates* through the sequence `animals`, and for each iteration:

- ▶ the next value in the sequence is assigned to variable `animal`
- ▶ the loop body is executed

Iterating over integers

Often we need to iterate over a sequence of integers:

```
>>> for i in [0, 1, 2, 3, 4, 5]:  
    print("the square of", i, "is", i**2)
```

the square of 0 is 0

the square of 1 is 1

the square of 2 is 4

the square of 3 is 9

the square of 4 is 16

the square of 5 is 25

Iterating over integers with range

The expression `range(n)` can be used to iterate over a sequence of increasing integer values up to (but not including) `n`:

```
>>> for i in range(6):  
        print("the square of", i, "is", i**2)
```

```
the square of 0 is 0
```

```
the square of 1 is 1
```

```
the square of 2 is 4
```

```
the square of 3 is 9
```

```
the square of 4 is 16
```

```
the square of 5 is 25
```

The range function

```
range(start: int = 0, stop: int, step: int = 1) -> Sequence[int]
```

iterates over integers from start to stop (but not including stop) in steps of step. start is optional (defaults to 0) and step is optional (defaults to 1).

- ▶ The `range` function returns a (lazy) sequence
- ▶ Sequences can be used in a for loop (and in many other places where a sequence is needed)
- ▶ You can convert a `range` expression into a list:

```
>>> list(range(6))
```

```
[0, 1, 2, 3, 4, 5]
```

Iterating over sequences

```
>>> for i in [0, 3, 4, 19]:  
    print(i)
```

```
0  
3  
4  
19
```

```
>>> for i in range(5):  
    print(i)                # range expressions  
                            # are sequences
```

```
0  
1  
2  
3  
4
```

```
>>> for letter in "Hello_World": # strings
      print(letter)             # are sequences
```

```
H
e
l
l
o

W
o
r
l
d
```

Another iteration example

This example generates a list of numbers often used in hotels to label floors ([more info](#))

```
>>> def skip13(a: int, b: int) -> list[int]:
    """Returns a list of ints from a to b without 13"""
    result = []
    for k in range(a, b):
        if k == 13:
            pass                # do nothing
        else:
            result.append(k)
    return result
```

```
>>> skip13(1, 20)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19]
```

Exercise: range_double

Write a function `range_double(n: int) -> list[int]` that generates a list of numbers similar to `list(range(n))`. In contrast to `list(range(n))`, each value in the list should be multiplied by 2. For instance:

```
>>> range_double(4)
```

```
[0, 2, 4, 6]
```

```
>>> range_double(10)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

For comparison, the behaviour of `range`:

```
>>> list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

For loop – summary

- ▶ use for loops to iterate over sequences (such as lists or strings)
- ▶ can use `range` to generate sequences of integers
- ▶ actually possible to iterate over any `iterable` (no just sequences)

While loops

- ▶ Reminder: a for loop iterates over a given sequence or iterator
- ▶ A while loop iterates **while a condition is fulfilled**

Example

```
>>> x = 64
      while x > 10:
          x = x // 2
          print(x)
```

32

16

8

What are variables?

- ▶ Variables are **names** given to “objects”
- ▶ Variables can be assigned (initialized) **and then re-assigned** (there is no constant in Python)
- ▶ An object can be **immutable** (always the same value) or **mutable** (its value can change).
- ▶ For instance, an integer is an **immutable object** (hence it is a value) and a list is a **mutable object**.

Variables and mutable objects

```
>>> a = [0, 2, 4, 6] # bind name 'a' to list
```

```
>>> a # object [0,2,4,6]
```

```
[0, 2, 4, 6]
```

```
>>> b = a # bind name 'b' to the same
```

```
>>> b # list object
```

```
[0, 2, 4, 6]
```

```
>>> b[1] = 10 # modify second element (via b)
```

```
>>> b # show b
```

```
[0, 10, 4, 6]
```

```
>>> a # show a
```

```
[0, 10, 4, 6]
```

“id”, “==” and “is”

- ▶ Two objects a and b are the same object if they live in the same place in memory.
- ▶ Python provides the `id` function that returns the identity of an object (its “location” or its “memory address”).
- ▶ We check with `id(a) == id(b)` or `a is b` whether a and b are the same object.
- ▶ Two different objects can have the same value: we check with `==`.

Example 1

```
>>> a = [1, 2, 3]
```

```
>>> b = [1, 2, 3]
```

```
>>> id(a) # some location in memory
```

```
4493008000
```

```
>>> id(b) # another location in memory
```

```
4494273088
```

```
>>> a is b # i.e. not the same objects
```

```
False
```

```
>>> a == b # but carry the same value
```

```
True
```

Example 2

```
>>> a = [1, 2, 3]
```

```
>>> b = a          # b is reference to object of a
```

```
>>> a is b        # thus they are the same
```

True

```
>>> a == b        # the value is thus (of course) the same
```

True

Objects, values and types – summary

<https://docs.python.org/3/reference/datamodel.html>

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects.

Every object has an identity, a type and a value. An object's **identity** never changes once it has been created; you may think of it as the object's address in memory. The **'is'** operator compares the identity of two objects; the **id()** function returns an integer representing its identity.

An object's type determines the operations that the object supports (e.g., “does it have a length?”) and also defines the possible values for objects of that type. The **type()** function returns an object's type [...]. Like its identity, an object's **type** is also unchangeable.

The **value** of some objects can change. Objects whose value can change are said to be **mutable**; objects whose value is unchangeable once they are created are called **immutable**. [...] An object's mutability is determined by its type; for instance, numbers, strings and tuples are immutable, while dictionaries and lists are mutable.

Types affect almost all aspects of object behavior. Even the importance of object identity is affected in some sense: for immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed.

E.g., after `a = 1; b = 1`, `a` and `b` may or may not refer to the same object with the value one, depending on the implementation, but after `c = []; d = []`, `c` and `d` are guaranteed to refer to two different, unique, newly created empty lists.