Find and replace

Finding and replacing are two very frequent tasks. Knowing how to use them and how they work will help you to be more effective.

Activity 1 (Find)**.**

Goal: learn different ways to search with `Python`.

1. **The operator "in".**
   The easiest way to know if a substring is present in a string is to use the operator "`in`". For example, the expression:
   `"NOT" in "TO BE OR NOT TO BE"`
   is equal to "True" because the substring **NOT** is present in the sentence.
   Deduce a function `find_in(string,substring)` that returns "True" or "False", depending on whether the substring is (or is not) present in the chain.
2. **The method** `find()`**.**
   The `find()` method of `Python` is used as `string.find(substring)` and returns the position to which the substring was found.
   Test this on the previous example. What does the function return if the substring is not found?
3. **The method** `index()`**.**
   The `index()` method has the same utility, it is used in the form `string.index(substring)` and returns the position to which the substring was found.
   Test this on the previous example. What does the function return if the substring is not found?
4. **Your function** `find()`**.**
   Write your own function `myfind(string,substring)` which returns the starting position of the substring if it is found (and returns `None` if it is not).
   You are not allowed to use some functions of `Python`, you only have the right to test if two characters are equal.

Activity 2 (Replace)**.**

Goal: replace portions of text with others.

1. The `replace()` method is used in the form:
   `string.replace(substring,new_substring)`
   Each time the sequence `substring` is found in `string`, it is replaced by `new_substring`.
   Transform the sentence **TO BE OR NOT TO BE** into **TO BE AND NOT TO BE**, then into **TO HAVE AND NOT TO HAVE**.

2. Write your own function `myreplace()` which you will call in the following form:
   `myreplace(string,substring,new_substring)`
   and which only replaces the first occurrence of `substring` found. For example, `myreplace("ABBA","B","XY")` returns `"AXYBA"`.
   Hint. You can use your `myfind()` function from the previous activity to find the starting position of the sequence to replace.
3. Improve your function to build a function `replace_all()` which now replaces all occurrences encountered.

Lesson 1 (Regular expressions *regex*).
The ***regular expressions*** allow you to search for substrings with greater freedom: for example, you can allow a wildcard character or several possible choices for a character. There are many other possibilities, but we are only studying these two.

1. We allow ourselves a joker letter symbolized by a point "**.**". For example, if we look for the expression "**P.R**" then:
   - **PORK**, **EMPIRE**, **PURE**, **REPORT** contain this group (for example for **PORK** the point plays the role of **O**),
   - but not the words **CAR**, **POOR**, **RAP**, **PRICE**.
2. We are still looking for groups of letters, we now allow ourselves several options. For example "**[CT]**" means "**C** or **T**". Thus the letter group "**[CT]O**" corresponds to the letter group "**CO**" or "**TO**". This group is therefore contained in **TOTEM**, **COST**, **ACTOR** but not in **BLOCK** nor **VOTE**. Similarly "**[ABC]**" would mean "**A** or **B** or **C**".

We will use regular expressions through a command:
`python_regex_find(string,exp)`
whose function is defined below.

```
from re import *

def python_regex_find(string,exp):
    pattern = search(exp,string)
    if pattern:
        return pattern.group(), pattern.start(), pattern.end()
    else:
        return None
```

Program it and test it. It returns: (1) the found substring, (2) the start position and (3) the end position.

python : re.search() - python_regex_find()

Use: `search(exp,string)`
or `python_regex_find(string,exp)`
Input: a string `string` and a regular expression `exp`
Output: the result of the search (the substring found, its start position, its end position)

Example with `string = "TO BE OR NOT TO BE"`

- with `exp = "N.T"`, then `python_regex_find(string, exp)` returns `('NOT', 9, 12)`.
- with `exp = "B..O"`, the function returns `('BE O', 3, 7)` (the space counts as a character).
- with `exp = "[NM]O"`, the function returns `('NO', 9, 11)`.
- with `exp = "[BC]..O[RS]"`, the function returns `('BE OR', 3, 8)`.

Activity 3 (Regular expressions *regex*).

Goal: program the search for simple regular expressions.

1. Program your function `regex_find_wildcard(string,exp)` who is looking for a substring that can contain one or more wildcards ".". The function must return: (1) the found substring, (2) the start position and (3) the end position (as for the function `python_regex_find()` above).
2. Program your function `regex_find_choice(string,exp)` who is looking for a substring that can contain one or more choices contained in tags "[]". The function must return again: (1) the found substring, (2) the start position and (3) the end position.
   Hint. You can start by writing a function `all_choices(exp)` that generates all possibilities from `exp`. For example, if `exp = "[AB]X[CD]Y"` then `all_choices(exp)` returns the list formed of: `"AXCY"`, `"BXCY"`, `"AXDY"` and `"BXDY"`.

Lesson 2 (Replace 0 and 1 and start again!).
We consider a "sentence" composed of only two possible letters **0** and **1**. In this sentence we will search for a pattern (a substring) and replace it with another one.

Example.
Apply the transformation 01 → 10 to the sentence 10110.
We read the sentence from left to right, we find the first pattern 01 from the second letter, we replace it with 10:

$$1(01)10 \longmapsto 1(10)10$$

We can start again from the beginning of the sentence obtained, with always the same transformation 01 → 10:

$$11(01)0 \longmapsto 11(10)0$$

The pattern 01 no longer appears in the sentence 11100 so the transformation 01 →10 now leaves this sentence unchanged.
Let's summarize: here is the effect of the iterated transformation 01 → 10 in the sentence 10110:

$$10110 \longmapsto 11010 \longmapsto 11100$$

Example.
Apply the transformation 001 →1100 to the sentence 0011.
A first time:

$$(001)1 \longmapsto (1100)1$$

A second time:

$$11(001) \longmapsto 11(1100)$$

And then the transformation no longer modifies the sentence.

Example.
Let's see a last example with the transformation 01 →1100 for the starting sentence 0001:

$$0001 \longmapsto 001100 \longmapsto 01100100 \longmapsto 1100100100 \longmapsto \cdots$$

We can iterate the transformation, to obtain longer and longer sentences.

Activity 4 (Replacement iterations).

Goal: study some transformations and their iterations.

We consider here only transformations of the type $0^a1^b \to 1^c0^d$, i.e. a pattern with first 0's then 1's is replaced by a pattern with first 1's then 0's.

1. **One iteration.**
   Using your `myreplace()` function from the first activity, check the above examples. Make sure you replace only one pattern at each step (the leftmost one).
   Example: the transformation 01 → 10 applied to the sentence 101, is calculated by `myreplace("101","01","10")` and returns `"110"`.
2. **Multiple iterations.**
   Program a function `iterations(sentence,pattern,new_pattern)` that, from a sentence, iterates the transformation. Once the sentence is stabilized, the function returns the number of iterations performed and the resulting sentence. If the number of iterations does not seem to stop (for example when it exceeds 1000) then returns `None`.
   Example. For the transformation 0011 → 1100 and the sentence 00001101, the sentences obtained

are:

$$000011011 \underset{1}{\longmapsto} 001100011 \underset{2}{\longmapsto} 110000011 \underset{3}{\longmapsto} 110001100 \underset{4}{\longmapsto} 110110000 \longmapsto \cdots$$

For this example, the call to the `iterations()` function returns 4 (the number of transformations before stabilization) and `"110110000"` (the stabilized sentence).

3. **The most iterations possible.**
   Program a function `max_iterations(p,pattern,new_pattern)` which, among all the sentences of length p, is looking for one of those that takes the longest to stabilize. This function returns:
   - the maximum number of iterations,
   - a sentence that achieves this maximum,
   - and the corresponding stabilized sentence.

   Example: for the transformation **01** → **100**, among all the sentences of length $p = 4$, the maximum number of possible iterations is 7. Such an example of a sentence is **0111**, which will stabilize (after 7 iterations) in **11100000000**. So the command `max_iteration(4,"01","100")` returns:

   `7, '0111', '11100000000'`

   Hint. To generate all sentences with a length of p formed of **0** and **1**, you can consult the chapter "Binary II" (activity 3).

4. **Categories of transformations.**
   - **Linear transformation.** Experimentally check that the transformation **0011** →**110** is *linear*, i.e. for all sentences with a length of p, there will be at most about p iterations before stabilization. For example, for $p = 10$, what is the maximum number of iterations?
   - **Quadratic transformation.** Experimentally check that the transformation **01** →**10** is *quadratic*, i.e. for all sentences with a length of p, there will be at most about p^2 iterations before stabilization. For example, for $p = 10$, what is the maximum number of iterations?
   - **Exponential transformation.** Experimentally check that the transformation **01** →**110** is *exponential*, i.e. for all sentences with a length of p, there will be a finite number of iterations, but that this number can be very large (much larger than p^2) before stabilization. For example, for $p = 10$, what is the maximum number of iterations?
   - **Transformation without end.** Experimentally verify that for the transformation **01** →**1100**, there are sentences that will never stabilize.