Make a 4-character string, and assign it to a name <= esto es Markdown

```
In [ ]: # Make a 4-character string, and assign it to a name <= esto es un comentario en Python
        S = 'Spam'
        print(S)
```

```
Spam
```

Length

```
In [ ]: len(S)
```

```
Out[ ]: 4
```

The first item in S, indexing by zero-based position

In Python, indexes are coded as offsets from the front, and so start from 0: the first item is at index 0, the second is at index 1, and so on.

```
In [ ]: S[0]
```

```
Out[ ]: 'S'
```

The second item from the left

```
In [ ]: S[1]
```

```
Out[ ]: 'p'
```

```
In [ ]: # Si te sales de los límites
        S[8]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-34-2e554675c715> in <module>
      1 # Si te sales de los límites
----> 2 S[8]

IndexError: string index out of range
```

The last item from the end in S

In Python, we can also index backward, from the end—positive indexes count from the left, and negative indexes count back from the right.

```
In [ ]: S[-1]

Out[ ]: 'm'
```

The second-to-last item from the end

```
In [ ]: S[-2]

Out[ ]: 'a'
```

Backus-Naur

Negative indexing, the hard way. Expresiones en Python <=> resolver por Backus-Naur

```
In [ ]: S[len(S) - 1]
```

```
Out[ ]: 'm'
```

Slice of S

```
In [ ]: # from offsets 1 through 2 (not 3)
        S[1:3]
```

```
Out[ ]: 'pa'
```

```
In [ ]: # Everything past the first (1:len(S))
        S[1:]
```

```
Out[ ]: 'pam'
```

```
In [ ]: # S itself hasn't changed
        S
```

```
Out[ ]: 'Spam'
```

```
In [ ]: # Everything but the last
        S[0:3]
```

```
Out[ ]: 'Spa'
```

```
In [ ]: # Same as S[0:3]
        S[:3]
```

```
Out[ ]: 'Spa'
```

```
In [ ]: # Everything but the last again, but simpler (0:-1)
        S[:-1]
```

```
Out[ ]: 'Spa'
```

```
In [ ]: # All of S as a top-level copy (0:len(S))
        S[:]
```

```
Out[ ]: 'Spam'
```

Concatenación y repetición

```
In [ ]: S + 'eggs'
```

```
Out[ ]: 'Spameggs'
```

```
In [ ]: # S is unchanged
        S
```

```
Out[ ]: 'Spam'
```

```
In [ ]: # Repetition
        S * 6
```

```
Out[ ]: 'SpamSpamSpamSpamSpamSpam'
```

Polimorfismo

The plus sign (+) means different things for different objects: addition for numbers, and concatenation for strings. This is a general property of Python called polymorphism. The meaning of an operation depends on the objects being operated on. As you'll see when we study **dynamic typing**, this polymorphism property accounts for much of the conciseness and flexibility of Python code. Because types aren't constrained, a Python-coded operation can normally work on many different types of objects automatically, as long as they support a compatible interface (like the + operation here).

Inmutabilidad

Immutable objects cannot be changed. Every object in Python is classified as either immutable (unchangeable) or not. In terms of the core types, **numbers, strings, and tuples are *immutable*; lists, dictionaries, and sets are *mutable***.

```
In [ ]: S[0] = 'z'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-40-0db6cb5bde2e> in <module>
----> 1 S[0] = 'z'

TypeError: 'str' object does not support item assignment
```

We can run expressions to make new objects

```
In [ ]: S = 'z' + S[1:]
        S
```

```
Out[ ]: 'zpam'
```

Strings y listas

Propiedades de los objetos

```
In [ ]: E = 'egss'
        L = list(E)
        print(L)
        L[0] = 'z'
        L
```

```
['e', 'g', 's', 's']
```

```
In [ ]: ''.join(L)
        # L es ['z', 'g', 's', 's']

Out[ ]: 'zgss'
```

Type-Specific Methods

Find

```
In [ ]: # Find the offset of a substring in S
        S = 'SpamEggsSpam'
        S.find('pa')

Out[ ]: 1

In [ ]: S.find('pa', 3)

Out[ ]: 9

In [ ]: S.find(S)

Out[ ]: 0

In [ ]: # Si no existe el caracter => -1
        S.find('z')

Out[ ]: -1
```

Replace

In [ ]:
```
S.replace('Eggs', 'Bacon')
# S es inmutable! S = 'SpamEggsSpam'
```
Out[ ]:
```
'SpamBaconSpam'
```

Mayúsculas

In [ ]:
```
S.upper()
# S es inmutable! S = 'SpamEggsSpam'
```
```
SpamEggsSpam
```

In [ ]:
```
print("S is alpha", S.isalpha())
print("S is digit", S.isdigit())
```
```
S is alpha True
S is digit False
```

Split

In [ ]:
```
S = 'spams-spam-eggs-spam-bacon'
S.split('-')
```
Out[ ]:
```
['spams', 'spam', 'eggs', 'spam', 'bacon']
```

Rstrip

Remove whitespace characters on the right side

In [ ]:
```
line = 'aaa\t,bbb\t,\nccccc,\tdd\n'
print(line)
```
```
aaa	,bbb	,
ccccc,	dd
```

```
In [ ]: line.rstrip()
```

```
Out[ ]: 'aaa\t,bbb\t,\ncccccc,\tdd'
```

```
In [ ]: line.lstrip()
```

```
Out[ ]: 'aaa\t,bbb\t,\ncccccc,\tdd\n'
```

```
In [ ]: # Combine two operations
        line.rstrip().split(',')
```

```
Out[ ]: ['aaa\t', 'bbb\t', '\ncccccc', '\tdd']
```

Formatting

Strings also support an advanced substitution operation known as **formatting**

```
In [ ]: # # Formatting expression
        '%s, eggs, and %s' % ('spam', 'SPAM!')
```

```
Out[ ]: 'spam, eggs, and SPAM!'
```

```
In [ ]: # Formatting method
        '{}, eggs, and {}'.format('spam', 'SPAM!')
```

```
Out[ ]: 'spam, eggs, and SPAM!'
```