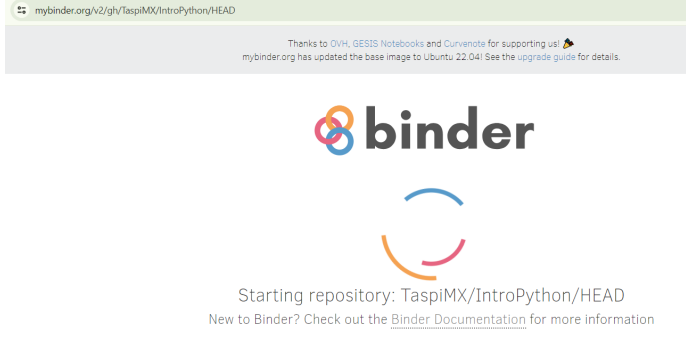
Empecemos a Programar!

Entrar al ambiente

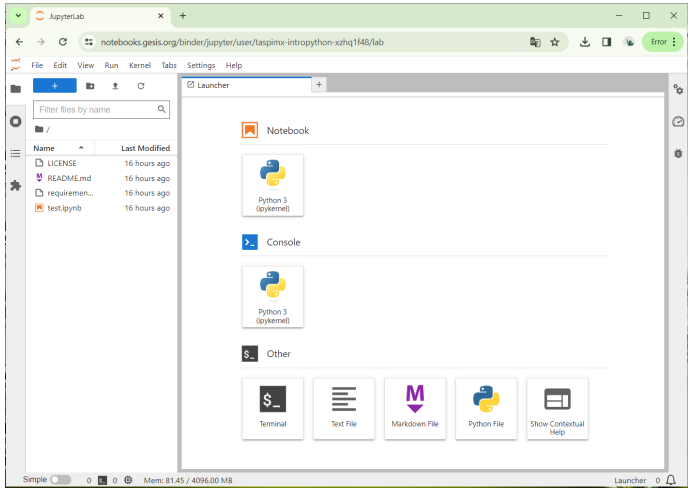
1. Entrar al ambiente (puede tardar ~5min par de minutos en cargar)

URL: https://t.ly/s_ySZ

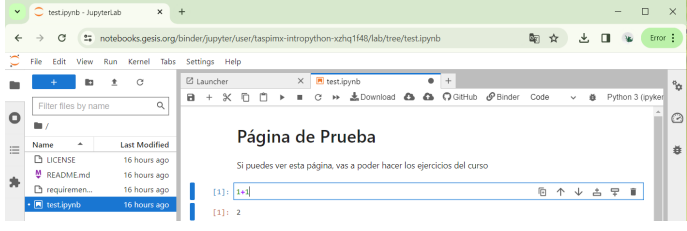


Entrar al ambiente

2. Pantalla del ambiente MyBinder



Abrir el cuaderno de trabajo "01_ejercicios.ipynb"



> Ejercicio 1.1.: Realiza una operación aritmética sencilla:

Posiciona el cursor en la siguiente celda y teclea una suma de dos números.

Posteriormente ejecútalo con `CONTROL y ENTER` simultáneamente

In [ ]:
```
1+2
```

Componentes básicos del lenguaje

- Variables
- Tipos de datos
- Palabras reservadas

Variables

- Una variable puede conceptualizarse como una caja, la cual "guarda" un valor
- Las variables se definen de forma dinámica:
    - no se tiene que especificar cuál es su tipo de antemano
    - puede tomar distintos valores en el programa, incluso de un tipo diferente al que tenía previamente
- Los nombres de variables pueden contener números, letras y guión bajo pero deben comenzar con una letra
- Se usa el símbolo = para asignar valores.

Reglas para crear un nombre de variable

- Los nombres de las variables deben comenzar con una letra (a-z, A-Z) o un guión bajo (_).
- El resto del nombre de la variable puede contener letras, números y guiones bajos.
- Los nombres de las variables son sensibles a mayúsculas y minúsculas, lo que significa que nombre, Nombre y NOMBRE se considerarán como tres variables diferentes
- . Los nombres de las variables no pueden comenzar con un númer
- o. No se pueden utilizar palabras clave reservadas de Python como nombres de variables. Algunas palabras clave reservadas incluyen if, else, for, while, def, class, entre ot
- Ejemplos:
    - edad = 18
    - ciudad = "CDMX"
    - cp = "52143"
    - tasa = 11.25ras.

Palabras reservadas que no pueden usarse en nombre de variables

- `and`
- `as`
- `assert`
- `async`
- `await`
- `break`
- `class`
- `continue`
- `def`
- `del`
- `elif`
- `else`
- `except`
- `False`
- `finally`
- `for`
- `from`
- `global`
- `if`
- `import`
- `in`
- `is`
- `lambda`
- `None`
- `nonlocal`
- `not`
- `or`
- `pass`
- `raise`
- `return`
- `True`
- `try`
- `while`
- `with`
- `yield`

La asignación

La *asignación* es una sentencia simple:

- Enlazan (*bind*) una variable con un valor, o bien modifican una variable (modificable)
- Forma: `variable = expresión`
- La variable *apuntará* al resultado de evaluar la expresión

In [ ]:
```
x = 3 + 4
print(x)
```

Ejercicio 1.2.: Realiza una operación aritmética sencilla, guardando los números en variables

Posiciona el cursor en la siguiente asigna los valores 1 y 2 a las variables `a` y `b` respectivamente. Posteriormente ejecútalo con `CONTROL y ENTER` simultáneamente

In [ ]:
```
a =
b =
print(a+b)
```

Ejercicio 1.3.: Calcula el interés simple de un crédito de $1,000 pesos, por una tasa de 11.00% y un plazo de 90 días. La fórmula es:

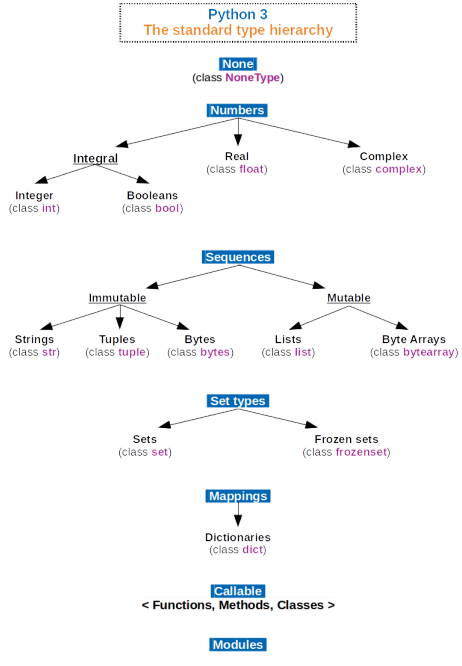
$$interes = capital * tasa/360 * plazo$$

Posiciona el cursor en la siguiente asigna los valores de las variables y la fórmula. Posteriormente ejecútalo con `CONTROL y ENTER` simultáneamente

In [ ]:
```
capital =
tasa =
plazo =
interes =
print()
```

Tipos

Todo objeto en Python de datos tiene un tipo asociado.



Ejemplos

Tipo	Clase	Notas	Ejemplo
str	Cadena en determinado formato de codificación (UTF-8 por defecto)	Inmutable	'Cadena'
bytes	Vector o array de bytes	Inmutable	b'Cadena'
list	Secuencia	Mutable, puede contener objetos de diversos tipos	[4.0, 'Cadena', True]
tuple	Secuencia	Inmutable, puede contener objetos de diversos tipos	(4.0, 'Cadena', True)
set	Conjunto	Mutable, sin orden, no contiene duplicados	{4.0, 'Cadena', True}
dict	Diccionario	Grupo de pares clave:valor	{'key1': 1.0, 'key2': False}
int	Número entero	Precisión arbitraria	42
float	Número decimal	Coma flotante de doble precisión	3.1415927
complex	Número complejo	Parte real y parte imaginaria j.	(4.5 + 3j)
bool	Booleano	Valor booleano (verdadero o falso)	True o False

Tipos de datos incorporados (*built-in*)

Son tipos de datos disponibles de inicio

- Suelen ser más eficientes que los definidos por los usuarios
- Extensibles (por nuevas *clases*)

Estos tipos son (entre otros):

- Números: `int`, `float`, `complex`
- Booleans: `False`, `True`
- Nulo: `None`
- Strings (`str`): `"abcdef"`
- Secuencias (listas, tuplas, sets...): `[0, 'a', 2]`
- Diccionarios (mapas): `{'a': 1, 'b': 3}`
- Otros: secuencias binarias (*bytes*), ficheros, clases, módulos, funciones, tipos...

In [ ]:
```
type(3)
```

In [ ]:
```
type(3.0)
```

In [ ]:
```
type('3')
```

In [ ]:
```
type( [0,1,2,3] )
```

Operaciones con diferentes tipos

Por lo general, no es posible realizar operaciones con diferentes tipos, al menos que la función específica lo admita.

- En general, no hay conversión de tipos automático (solo en casos concretos).

In [ ]:
```
"Resultado: " + str(25)
```

Ejercicio 1.4.: Asigna el valor de 100 a la variable `capital` y posteriormente imprime el mensaje: "El capital es de $100":

Posiciona el cursor en la siguiente celda y asigna los valores de las variables y la funcion print. Posteriormente ejecútalo con `CONTROL y ENTER` simultáneamente

In [ ]:
```
capital =
print("El capital es " +   )
```

La operacion `+` acepta valores del mismo tipo. Para poder `concatenar` un texto con un número, hay que utilizar la función `str()` para convertir el número a texto.

str - Cadenas de Caracteres

Cadenas de caracteres se almacenan como *puntos de código* de Unicode (es decir, sin una codificación concreta).

- Por defecto, se usa la codificación *UTF-8* para operaciones de entrada/salida.

En su manera más simple (y habitual), se crean con literales, usando `'` o `"`, de varias formas:

In [ ]:
```
print("Hola")
```

- Se puede accesar un caracter del string utilizando `[]`
- Importante resaltar que el primer caracter está en la posición 0
- También se puede accesar un corte del string utilizando `[pos inicial:pos final]`, considera que la posición final no se incluye.

In [ ]:

```
saludo = "Hola mundo!"
print( saludo[5] )
print(saludo[5:8] )
print( saludo[3:] )
```

Ejercicio 1.5.: Asigna tu nombre a la variable nombre y despliega el valor utilizando print.

Posiciona el cursor en la siguiente celda y asigna los valores de las variables y la funcion print. Posteriormente ejecútalo con `CONTROL y ENTER` simultáneamente

In [ ]:
```
nombre = ""
print(nombre)
```

Ahora imprime el tercer caracter de tu nombre, utilizando corchetes `[2]`

In [ ]:
```
print(nombre[2])
```

Ahora imprime del tercer al sexto caracter, utilizando corchetes y los dos puntos para definir un rango `[2:5]`. (NOTA: por convención, el elemento de la derecha no se incluye).

In [ ]:
```
print(nombre[2:5])
```

- Como sus caracteres pueden recorrerse uno a uno (son *iterables*), los strings también son secuencias.
    - Eso implica que aceptan muchos métodos de secuencias, y se pueden usar en un bucle `for` que veremos más adelante.

Concatenar cadenas de texto

- El operador `+` permite concatenar o unir cadenas de caracteres

In [ ]:
```
'ABC' + 'def'
```

In [ ]:
```
nombre = 'Juan'
print( " El nombre es: " + nombre )
```

Funciones de strings

- Los strings son objetos que tienen `metodos`, es decir, funciones incorporadas

In [ ]:
```
nombre = "python es lo máximo"
nombre.capitalize()
```

In [ ]:
```
nombre.upper()
```

Números

Funcionan como en otros lenguajes

- Enteros: `int`
- Decimal: `float`
- Complejos: `complex`

In [ ]:
```
print(type(3))
print(type(2.0))
print(type(2+5j))
```

Soportan las operaciones habituales:

Entre otras: + - * / % abs() ** //

In [ ]:
```
a = 5
b = 2
print(' a / b -->', a/b)
print('a // b -->', a//b)
print(' a % b -->', a%b)
```

list - Listas

En ciencia de datos, se deben manejar múltiples datos. Es poco práctico asignar una variable para cada punto.

Por ejemplo, las alturas de un grupo pueden ser

In [ ]:
```
altura1 = 1.73
altura2 = 1.68
altura3 = 1.71
altura4 = 1.89
altura5 = 1.73
```

Es muy difícil almacenar cientos o miles de datos utilizando variables separadas.

La solución es utilizar tipos que puedan guardar diferentes valores en una sóla variable.

La lista se representa por valores encerrados entre corchetes y separados por comas.

Por ejemplo, se pueden guardar todas las alturas del grupo en una sóla variable tipo `lista`:

In [ ]:
```
grupo = [1.73, 1.68, 1.71, 1.89, 1.73]
```

In [ ]:
```
print(grupo[0])
```

In [ ]:
```
print(grupo[2])
```

In [ ]:
```
print(grupo[-1])
```

Una lista puede contener valores de diferentes tipos.

In [ ]:
```
grupo2 = ["liz", 1.73, "juan", 1.68, "pedro", 1.71, "maria", 1.89, "helena", 1
```

In [ ]:
```
print( grupo2[0:2] )
```

```
[ inicio(incluido) : final(excluido) ]
```

Cambiar un elemento de una lista

In [ ]:
```
grupo2 = ["liz", 1.73, "juan", 1.68, "pedro", 1.71, "maria", 1.89, "helena", 1
print(grupo2)
```

- cambiar el nombre de 'pedro' por 'Peter'

Cambiar un elemento de una lista

In [ ]:
```
grupo2 = ["liz", 1.73, "juan", 1.68, "pedro", 1.71, "maria", 1.89, "helena", 1
print(grupo2)
```

- cambiar el nombre de 'pedro' por 'Peter'

In [ ]:
```
grupo2[4] = 'Peter'
print(grupo2)
```

Borrar un elemento de la lista

In [ ]:
```
grupo2 = ["liz", 1.73, "juan", 1.68, "pedro", 1.71, "maria", 1.89, "helena", 1
print(grupo2)
```

- borrar el nombre y la altura de 'Elena'

In [ ]:
```
del(grupo2[8:10])
print(grupo2)
```

Añadir un elemento a la lista

In [ ]:
```
grupo2 = ["liz", 1.73, "juan", 1.68, "pedro", 1.71, "maria", 1.89, "helena", 1
print(grupo2)
```

- Añadir a 'Mariana' con una altura de 1.80

In [ ]:
```
grupo2 = grupo2 + ['Mariana', 1.80]
print(grupo2)
```

También se puede realizar con la funcion .append()

Copiar repetidamente elementos

In [ ]:
```
print([1, 2] * 3)
```

Ejercicio 1.6.: Ejercicios con listas.

A partir de la siguiente lista, realiza los siguientes ejercicios:

In [ ]:

```
estados = ["Aguascalientes",
"Baja California",
"Baja California Sur",
"Campeche",
"Chiapas",
"Chihuahua",
"Ciudad de México",
"Coahuila",
"Colima",
"Durango",
"Estado de México",
"Guanajuato",
"Guerrero",
"Hidalgo",
"Jalisco",
"Michoacán",
"Morelos",
"Nayarit",
"Nuevo León",
"Oaxaca",
"Puebla",
"Querétaro",
"Quintana Roo",
"San Luis Potosí",
"Sinaloa",
"Sonora",
"Tabasco",
"Tamaulipas",
"Tlaxcala",
"Veracruz",
```

```
"Yucatán",
"Zacatecas"]
```

- ¿Cuál es el número de estados? Utiliza la funcion `len()`

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

- Cambia el "Estado de México" por solamente "México", hazlo directamente utilizando la posición del estado [] y posteriormente imprime la lista para revisar el resultado

In [ ]:
```

```

- Ordena la lista en orden alfabético, utiliza el método `.sort()` e imprime el resultado.

In [ ]:
```
estados.sort()
print(estados)
```

Ranges

Secuencias inmutables y ordenadas de enteros.

- Muy usados p. ej. en bucles (`for`)
- no almacena todos los valores que contiene, sino que los *genera* sobre la marcha, conforme se le solicitan.
    - `range` solo almacena *inicio*, *final*, *paso*.

In [ ]:
```
for val in range(5):  print(val)
```

In [ ]:
```
r = range(0, 10, 2)
print(r)
## el objeto no imprime los valores, hay que hacerlo manualmente
```

In [ ]:
```
for x in r:  print(x)
```

In [ ]:
```
list(r)
```

Ejercicio 1.7.: Utilizar rangos

Crea un rango iniciando en 1 y terminando en 20 y posteriormente imprímelos con un ciclo `for`.

- `range` solo almacena *inicio*, *final*, *paso*.

In [ ]:
```
rango = range(20)

for x in rango: print(x)
```

In [ ]:
```
rango = range(20)

for x in rango: print(x)
```

Boolean

Los `booleans` solo incluyen a dos valores: las contantes predefinidas: `True` y `False`.

Estos dos valores pueden ser el resultado de evaluar:

- Las propias contantes en un programa: `True` y `False`
- Una operación lógica: `or`, `and`, `not`
- Una comparación: `<`, `<=`, `>`, `>=`, `==`, `!=`, `is`, `is not`
- Evaluación de cualquier objeto python
    - Por defecto, la evaluación es a `True`
    - Excepto: `False`, `None`, `0`, secuencias vacías, objetos con un método especial `__bool__` que devuelva `False`

In [ ]:

```
if 4 == 4: print('4==4', '\t-->  Verdadero')
else:      print('4==4', '\t-->  Falso')

if 4 == 5: print('4==5', '\t-->  Verdadero')
else:      print('4==5', '\t-->  Falso')
```

Expresiones y sentencias

Una *expresión* es una porción de código que se evalúa a un valor.

- Una expresión simple es una constante (numérica o string), o una variable: `3`, `'a'`, `var`
- Las expresiones se relacionan por medio de operadores, creando expresiones más complejas: `3 + 2`, `['a', 'b']`, `mi_funcion(var)`

Una *sentencia* es una instrucción que python puede interpretar (ejecutar)

- Las expresiones forman parte de sentencias
- Existen sentencias simples y compuestas (involucrando varias sentencias simples)

Operadores

Operación	Sintaxis	Función
Adición	a + b	add(a, b)
Concatenación (cadenas)	seq1 + seq2	concat(seq1, seq2)
Chequeo de pertenencia	obj in seq	contains(seq, obj)
División	a / b	truediv(a, b)
División	a // b	floordiv(a, b)
Exponenciación	a ** b	pow(a, b)
Identidad	a is b	is_(a, b)
Identidad	a is not b	is_not(a, b)
Asignación indexada	obj[t;= b	ge(a, b)
Ordenado	a > b	gt(a, b)

Operación	Sintaxis	Función
setitem(obj, k, v)		
Indexado	obj[k]	getitem(obj, k)
Módulo	a % b	mod(a, b)
Multiplicación	a * b	mul(a, b)
Multiplicación de matrices	a @ b	matmul(a, b)
Negación (aritmética)	- a	neg(a)
Negación (lógica)	not a	not_(a)
Positivo	+ a	pos(a)
Segmentación	seq[i:j]	getitem(seq, slice(i, j))
Sustracción	a - b	sub(a, b)
Ordenado	a < b	lt(a, b)
Ordenado	a <= b	le(a, b)
Igualdad	a == b	eq(a, b)
Diferencia	a != b	ne(a, b)
Ordenado	a >= b	ge(a, b)
Ordenado	a > b	gt(a, b)

Jerarquía de las operaciones

1. ( )
2. a ^ n
3. a * b
4. a + b

Si hay operadores del mismo nivel, se resuelve de izquierda a derecha

Ejercicio 1.7.: Ejecutar las siguientes operaciones aritméticas

Entes de ejecutar estas operaciones, calcula mentalmente el resultado. Considera la `precedencia` de los operadores.

In [ ]:
```
5 - 3 - 2 * 2
```

In [ ]:
```
5 - 2 ^ 2
```

In [ ]:
```

```

Condicionales y control de flujo

La sentencia IF

Es una sentencia compuesta de control de flujo; es decir, para ejecutar unas instrucciones, u otras, en función de alguna condición.

Su forma más básica es:

```
if <condicion>:  sentencia
```

O bien:

```
if <condicion>:
        sentencia
        ...
```

Más generalmente, para especificar alternativas:

```
if <condicion>:
        do something

    elif <otra condicion>:
        do something else
    ...

    else:
        do this if nothing above matched
```

In [ ]:

```
if 2 == 3:
    print('2 es 3')
else:
    print('No! 2 no es 3')
```

In [ ]:

```
edad = 25

print("La persona es")
if edad < 18: # el if termina con : para indicar donde acaba la condición
    # el print va indentado con 4 espacios para indicar que está dentro del
    # cuerpo del if
    print("Menor")
else:
    #Lo mismo con este print
    print("Mayor")

print("De edad")
```

Ejercicio 1.9.: Escribir una sentencia IF, que compruebe una variable numérica `x`.

- Si `x` es positiva, mostrará "POSITIVO"
- Si `x` es negativa, mostrará "NEGATIVO"

Probarlo para varios valores de `x`, incluido 10.

In [ ]:

```
x = 10


if  :

else:
```

Ahora, agrega una condición adicional: Si `x` es cero, mostrará "ZERO"

In [ ]:
```
x = 10


if  :

else:
```

Ahora, agrega una condición adicional: Si `x` es cero, mostrará "ZERO"

For

La sentencia for de Python itera sobre los ítems de cualquier secuencia (una lista o una cadena de texto), en el orden que aparecen en la secuencia. Por ejemplo, el siguiente código muestra cada palabla de la lista con su longitud:

In [ ]:
```
# Longitud de las siguientes cadenas:
words = ['gato', 'ventana', 'computadora']
for w in words:
    print(w, len(w))
```

Los ciclos for tienen la sintaxis `for variable in lista:`. En este caso, variable es la variable que va a ir cambiando, y lista es una lista de python (o un iterable que es parecido)

Ejercicio 1.10.: Sumar los elementos de una lista

Utilizar un ciclo `for`

In [ ]:

```
alturas = [1.73, 1.68, 1.71, 1.89, 1.73]
# creo una variable inicial con valor cero
suma = 0
# recorro cada uno de los elementos de la lista
for x in alturas:
    # sumo cada numero en la variable suma
    suma += x

# al final, la variable suma debe tener el total
print(suma)
```

Utilizando el ciclo for para recorrer un string

In [ ]:
```
v1 = "Este es mi string"
print("Longitud de v1:", len(v1))
```

In [ ]:
```
v1[8]
```

In [ ]:
```
for char in v1:
    print(char)
```

Funciones

Código reutilizable Resuelve una actividad en particular Hay innumerables funciones disponibles en librerias. Ejemplo:

In [ ]:
```
grupo = [1.73, 1.68, 1.71, 1.89, 1.73]

max(grupo)
```

In [ ]:
```
sum(grupo) / len(grupo)
```

- `sum()`, `max()`, `len()` son ejemplos de funciones incorporadas al lenguaje.

Otras funciones

In [ ]:
```
print( round(3.14159,2) )
```

In [ ]:
```
print( len("Curso de Python") )
```

In [ ]:
```
grupo = [1.73, 1.68, 1.71, 1.89, 1.73]
grupo.count(1.73)
```

Función personalizada

Es posible construir una función personalizada para evitar la repetición de código.

Por ejemplo, tomemos el cálculo del Indice de Masa Corporal.

$$IMC = \frac{peso}{altura^2}$$

Podemos hacerlo de forma manual, por ejemplo, para una persona de peso 71kg y altura 1.62m:

In [ ]:
```
print (71 / 1.62**2)
```

In [ ]:
```
También podemos crear una fórumula personalizada:
```

In [ ]:
```
def IMC ( peso, altura):
    return(peso / altura**2)
```

In [ ]:
```
print(IMC( 71, 1.62) )
```

In [ ]:
```
print( IMC(68, 1.65) )
```

Ejercicio 1.11.: Crear una funcion para calcular el interés simple:

$interes = capital * tasa/360 * plazo$

La funcion debe recibir 3 datos: capital en pesos, tasa en decimal, plazo en días, y regresar un sólo valor correspondiente interés en pesos.

In [ ]:
```
def interes():

    return
```

Prueba la función con los siguientes datos:

- Capital: $1,000, Tasa: 11.50%, Plazo: 91 días
- Capital: $5,000, Tasa: 9.50%, Plazo: 182 días

In [ ]:
```
print( interes() )
```

Concepto de Objeto

En python cualquier *dato* (o *valor*) es un objeto:

- Números, listas, funciones, clases...

Los objetos tienen atributos (miembros):

- **Datos**: Información útil para el objeto
- **Métodos** (funciones): Operaciones sobre el propio objeto

Los atributos son accesibles con la notación `objeto.atributo`

Podemos inspeccionar los atributos de un objeto con los comandos `dir`, `help`

In [ ]:
```
a = "Hola Mundo"
dir(a)
```

In [ ]:
```
help(a.split)
```

In [ ]:
```
# por ejemplo, si queremos separar la siguiente serie de tasas de un archivo s
tasas = "11.30,11.28,11.34,11.25"
tasas.split(sep=",")
```