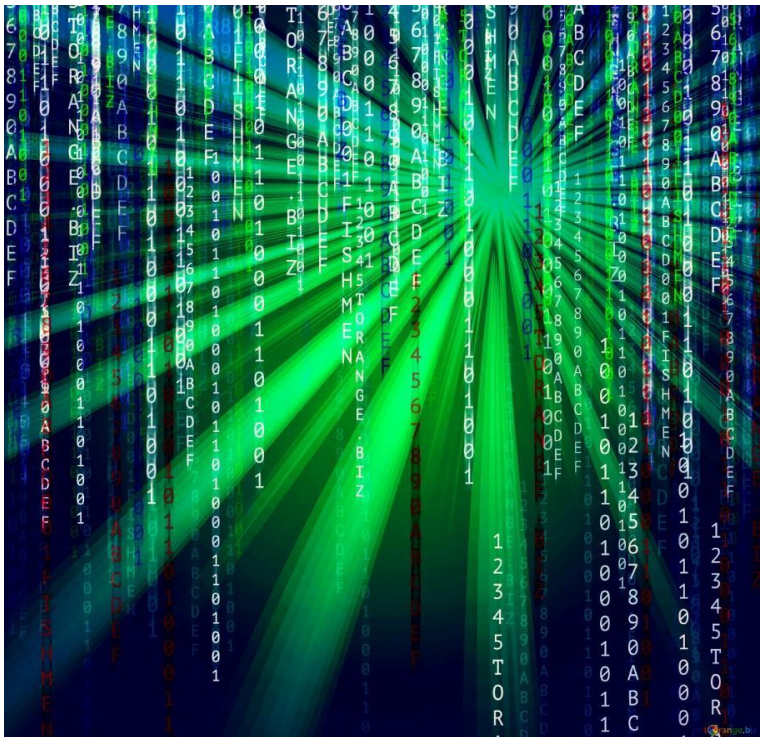
Clase 4

Programar en Python

Sintaxis de Python: Comenzando a escribir código.

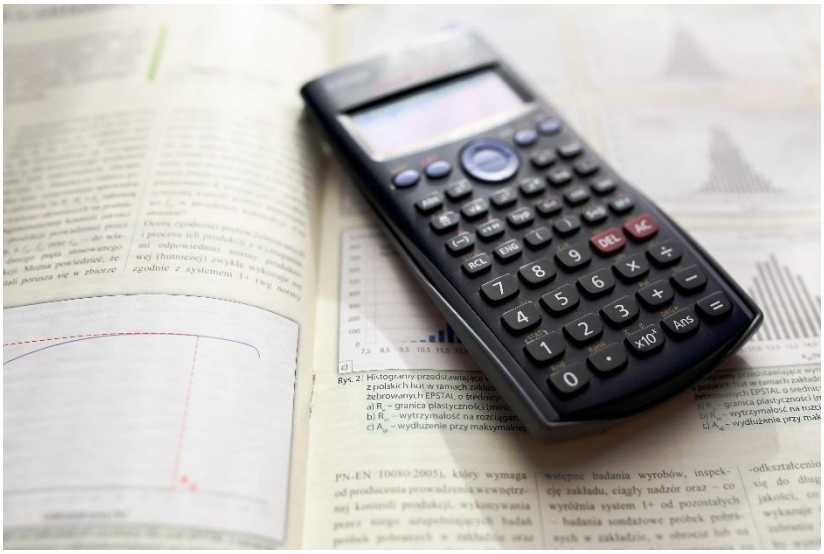


A lo largo de esta clase empezaremos a escribir código y a familiarizarnos con el uso de python, mayormente mediante ejemplos de códigos. Iremos incrementando la dificultad de los mismos conforme vayamos avanzando.

Ejemplos en Python

En los siguientes ejemplos que mostraremos, las entradas y salidas son distinguidas por la presencia o ausencia de los prompts (>>> y ...): para reproducir los ejemplos es necesario escribir todo lo que se encuentre después del prompt, cuando este aparezca; las líneas que no comiencen con el prompt son las salidas del intérprete. Además, hay que tener en cuenta que el prompt secundario que aparece por sí sólo en una línea de un ejemplo significa que es necesario escribir una línea en blanco; esto es usado para terminar un comando multilínea, como ya hemos mencionado con anterioridad.
Muchos de los ejemplos incluyen comentarios. Ya que los comentarios son para aclarar código y no son interpretados por Python, pueden omitirse cuando se testeen los ejemplos.

Utilizar Python para operaciones matemáticas



Similar a lo que hemos realizado para probar la correcta instalación de Python en la clase 2, probaremos algunos comandos simples a partir de un ejemplo. Para eso es necesario iniciar el intérprete.

Números

A lo largo de esta clase, veremos una serie de particularidades del uso de números en Python. Para eso, realizaremos algunos ejemplos básicos utilizando Python como si fuese una calculadora.
El intérprete puede actuar como una calculadora; es decir, es posible ingresar una expresión matemática y el intérprete nos devolverá los valores resultantes de dicha operación. Intuitivamente ya sabíamos esto luego de los breves testeos que realizamos al instalar python en nuestro sistema.
La sintaxis es sencilla: los operadores '+, -, * y /' funcionan como en la mayoría de los lenguajes (por ejemplo, Pascal o C); los paréntesis '()' pueden ser usados para agrupar. Por ejemplo:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # la división siempre retorna un número de punto flotante
1.6
```

Hemos mencionado en clases anteriores que Python es un lenguaje "no tipado", esto puede malinterpretarse, pensando que Python no maneja diferentes tipos de datos, lo cual no es cierto. Python es un lenguaje de alto nivel, por lo que tiene muchas características que le facilitan la vida al programador. La elección del tipo de dato es una de ellas. Para declarar una

variable en Python, basta con escribir su nombre y asignarle un valor. Al realizar este paso le estamos indicando en forma implícita el tipo de dato que contendrá. No obstante, tenemos el poder de cambiar en forma dinámica el tipo de dato de dicha variable, realizando una nueva asignación involucrando un dato de otra índole.
Veamos cómo maneja Python los tipos de datos numéricos:
Los números enteros (por ejemplo 2, 4, 20) son de tipo int, aquellos con una parte fraccional (por ejemplo 5.0, 1.6) son de tipo float.
La división (/) siempre retorna un punto flotante. Si nos interesa quedarnos solo con la parte entera de una división, lo que se conoce como *floor division,* podemos utilizar el operador //; si lo que queremos es calcular el resto, utilizaremos %:

```
>>> 17 / 3  # la división clásica retorna un punto flotante
5.666666666666667
>>>
>>> 17 // 3  # la división entera descarta la parte fraccional
5
>>> 17 % 3  # el operando % retorna el resto de la división
2
>>> 5 * 3 + 2  # resultado * divisor + resto
17
```

Con Python, es posible usar el operador '**' para calcular potencias:

```
>>> 5 ** 2  # 5 al cuadrado
25
>>> 2 ** 7  # 2 a la potencia de 7
128
```

El signo igual (=) es usado para asignar un valor a una variable. Por consiguiente, no se mostrará ningún resultado antes del próximo prompt:

```
>>> ancho = 20
>>> largo = 5 * 9
>>> ancho * largo
900
```

Si una variable no está "definida" (con un valor asignado), intentar usarla producirá un error:

```
>>> n  # tratamos de acceder a una variable no definida
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Hay soporte completo de punto flotante; operadores con operando mezclados convertirán los enteros a punto flotante:

```
>>> 4 * 3.75 - 1
14.0
```

En el modo interactivo, la última expresión impresa es asignada a la variable _. O sea que podemos seguir calculando a partir del último resultado obtenido teniendo en cuenta esa variable:

```
>>> impuesto = 12.5 / 100
>>> precio = 100.50
>>> precio * impuesto
12.5625
>>> precio + _
113.0625
>>> round(_, 2)
113.06
```

Esta variable debería tomarse como variable de solo lectura. Si bien es posible asignarle un valor, esto no es recomendable, ya que, en realidad al asignarle un valor, no estaríamos utilizando dicha variable, sino creando otra variable local con el mismo nombre, enmascarando la variable que mencionamos inicialmente.
Además de int y float, Python maneja otros tipos de datos numéricos tales como Decimal y Fraction. Python también tiene soporte integrado para números complejos, y usa el sufijo j o J para indicar la parte imaginaria (por ejemplo 3+5j).

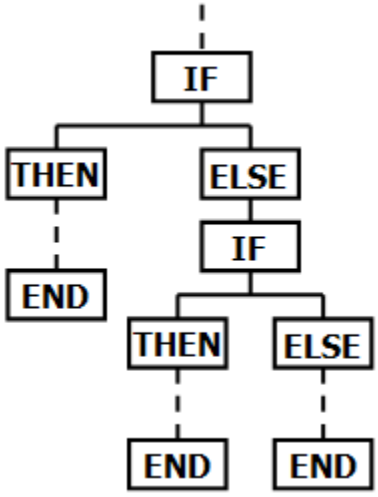
Realizando operaciones más complejas con números.

Hasta aquí hemos realizado algunas operaciones con números muy sencillas, con los mismos operandos que utilizaríamos en una calculadora científica y apenas algunos más específicos tales como quedarnos con la parte entera de un resultado en una división.

Pero ¿qué ocurriría si de repente quisiéramos calcular un resultado y en base a eso tomar una cierta decisión?.

Con las herramientas que conocemos hasta el momento, no podríamos realizarlo, por ese motivo introduciremos una serie de herramientas para control de flujo que nos permitirán realizar esto y mucho más.

La sentencia if



Supongamos que quiero realizar unos cálculos cuyo resultado no puede ser nunca negativo, es decir el resultado no me interesa si es menor a cero.
En ese caso, si pensáramos los pasos del código en forma de una lista, para poder trabajar con dicha condición la serie de pasos a seguir sería de la siguiente manera:

1. Introducir los valores a operar
2. Calcular la operación
3. ¿el resultado de la operación es menor a cero?
   a. Si es menor a cero: mostrar un cartel por pantalla comunicando el error
   b. Si no es menor a cero: mostrar el resultado de la operación por pantalla.
4. Fin

Lo que acabamos de resumir en unas líneas de texto, podría definirse como pseudo código y es de suma importancia y utilidad a la hora de diagramar nuestros programas.

¿Cómo se traduciría esto en lenguaje Python?, introduciendo la sentencia 'if'.

La sentencia IF nos da la posibilidad de plantear un interrogante y continuar el hilo del programa hacia la opción que cumpla con la condición que se ha planteado, para ello, veremos en el siguiente ejemplo tanto el uso de la sentencia 'IF' como la importancia de la Indentación para definir los bloques de acción.

'IF' viene acompañado de otra sentencia: 'ELSE', que actuará en el caso en que la condición que planteamos en 'IF' no se cumpla… en español sería 'Si' pasa esto entonces actúo de esta manera 'Si No se cumple' entonces actúo de esta otra manera.
El uso de 'ELSE' no es obligatorio, dependerá de la condición que estemos analizando, como veremos en otros ejemplos.

```
>>> valor1=4
>>> valor2=2
>>> result= valor1-valor2
2
>>> if result<0:
…       print('El resultado es negativo')
…   else:
…       print(result)
2
```

¿Qué ocurre si la decisión que tenemos que tomar no es binaria?.

No siempre nos alcanzará con dos opciones para definir una situación. Supongamos que en el código anterior nuestro resultado solo es útil si el número es mayor a 0 y menor a 100. En ese caso, introduciremos la sentencia 'elif', esta palabra reservada es una abreviación de 'else if' y se utiliza para simplificar el código.

```
>>> valor1=4
>>> valor2=2
>>> result= valor1-valor2
2
>>> if result<0:
…       print('El resultado es negativo')
…   elif result>100:
…       print('El resultado se encuentra fuera de rango')
…  else:
....      print(result)
```

En Python se utiliza esta secuencia de if/elif/else que reemplaza perfectamente a las sentencias switch/case que podemos encontrar en otros lenguajes de programación.

Veamos un resumen de todo lo dicho sobre estas sentencias condicionales en el siguiente video:
https://youtu.be/-kFBwApYVtA

La sentencia While

Otra sentencia de suma utilidad y que sin duda necesitaremos en nuestros códigos, es la sentencia 'While'.
Esta sentencia se utiliza para realizar una repetición de una acción mientras se cumpla con una condición explicitada.
Supongamos que queremos sumar todos los valores entre 1 y 10:

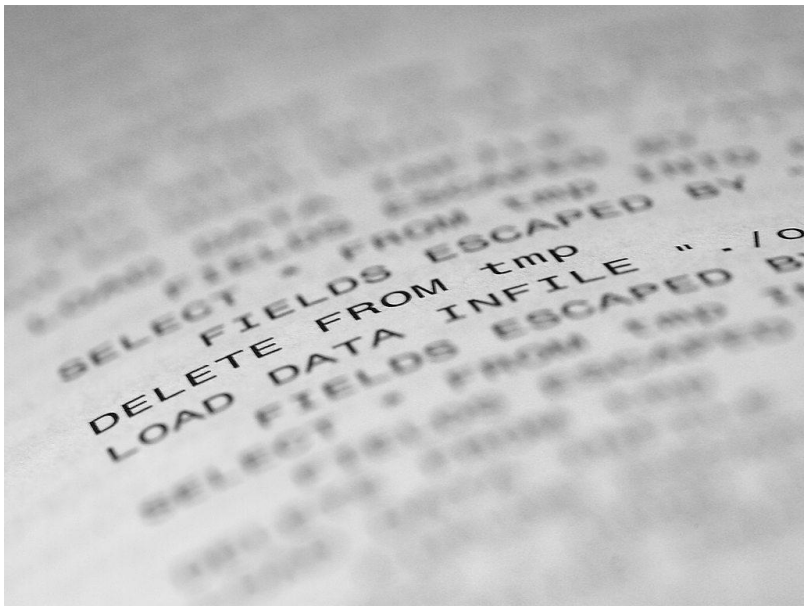
```
>>>b=1
>>>while b<10:
…     result=result+b
…     b=b+1
… print(result)
```

Analicemos un poco este código: la sentencia ‘while’ genera un bucle de código que se ejecutará mientras la condición se cumpla, en este caso mientras b valga menos de 10.
El bucle en cuestión estará conformado por las líneas que se encuentran indentadas de la misma manera, y en este caso vemos que son dos. Luego se imprime una única vez el resultado, ya que la indentación de dicho print nos da cuenta de que esa línea de código no pertenece al bloque ‘while’ y se ejecutará recién una vez finalizada la ejecución del mismo.

Veamos otro ejemplo y ampliemos un poco mas el uso de la sentencia while con el siguiente video:

https://youtu.be/YEWxlbffgxE

Otro tipo de datos: Cadenas de caracteres



Hasta el momento hemos visto cómo utilizar variables numéricas y algunas sentencias útiles para aplicar a nuestros códigos y poder realizar tareas más interesantes que cálculos que podríamos realizar con una calculadora.
Veamos ahora otro tipo de dato, que nos permitirá ampliar el horizonte de posibilidades en el mundo de Python.

Las cadenas de caracteres en su forma más sencilla son vectores de letras que definimos para formar un texto, podemos utilizarlas para almacenar mensajes que nos sirvan para darle información al usuario de nuestro código o podemos utilizarlas para almacenar datos que necesitemos procesar de alguna manera.

Las cadenas de texto pueden expresarse de diferentes maneras. Una forma de definirlas es encerrarlas en comillas simples o dobles.
Existen algunos caracteres especiales que no podrán ingresarse de manera directa en la cadena y para ellos será necesario contar con algún elemento que nos permita “escapar” de las comillas para poder introducir el caracter en cuestión. Para esos casos, se suele utilizar ‘\’.
Un ejemplo de cómo definir una cadena de caracteres:

```
>>> 'huevos y pan'  # comillas simples
'huevos y pan'
>>> 'doesn\'t'  # usa \' para escapar comillas simples…
"doesn't"
>>> "doesn't"  # ...o de lo contrario usa comillas doblas
"doesn't"
>>> '"Si," le dijo.'
'"Si," le dijo.'
>>> "\"Si,\" le dijo."
'"Si," le dijo.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Analicemos la segunda sentencia del ejemplo anterior, si no utilizáramos ‘\’ para escapar de las comillas simples, el intérprete entendería que la cadena de caracteres termina en el apóstrofe del texto y lo demás generaría un error al no entenderse como parte de la cadena.
Para entender mejor el texto que estamos introduciendo en la cadena, podemos valernos de la función print() y obtener una salida más legible, ya que nos mostrará el texto de la cadena tal cual lo vería el usuario final, sin las comillas ni los escapes para caracteres especiales:

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> print('"Isn\'t," she said.')
"Isn't," she said.
>>> s = 'Primera línea.\nSegunda línea.'  # \n significa nueva línea
>>> s  # sin print(), \n es incluído en la salida
'Primera línea.\nSegunda línea.'
>>> print(s)  # con print(), \n produce una nueva línea
Primera línea.
Segunda línea.
```

Si no queremos que los caracteres antepuestos por \ sean interpretados como caracteres especiales, podemos usar cadenas crudas agregando una r antes de la primera comilla:

```
>>> print('C:\algun\nombre')  # aquí \n significa nueva línea!
C:\algun
ombre
>>> print(r'C:\algun\nombre')  # nota la r antes de la comilla
C:\algun\nombre
```

Podemos escribir cadenas de texto con múltiples líneas. Para eso se suele utilizar triple comilla para encerrar al texto, pueden utilizarse comillas dobles o simples indistintamente.
Los finales de línea se agregan automáticamente, aunque es posible evitarlos agregando '\' al final de la línea.
Veamos un ejemplo:

```
print("""\
Uso: algo [OPTIONS]
    -h                  Muestra el mensaje de uso
    -H nombrehost           Nombre del host al cual conectarse
""")
```

produce la siguiente salida: (notar que la línea inicial no está incluída)

```
Uso: algo [OPTIONS]
    -h                  Muestra el mensaje de uso
    -H nombrehost           Nombre del host al cual conectarse
```

Una operación útil que podemos realizar al tener diferentes cadenas de caracteres es la concatenación, que consiste en pegar una cadena de caracteres a continuación de otra, esto se logra con el operador '+'. Además de concatenar, también es posible repetir una cadena de caracteres con el operador '*':

```
>>> # 3 veces 'un', seguido de 'ium'
>>> 3 * 'un' + 'ium'
'ununium'
```

Como podemos notar en el ejemplo anterior, debemos indicar cuantas veces queremos repetir o multiplicar la cadena de texto previo al uso del operador '*'.
Una particularidad de las cadenas literales, es decir aquellas que definimos encerrando entre comillas, es que, si las colocamos una al lado de otra, se concatenaran automáticamente, una particularidad que solo funciona con dos y solo dos literales y no sirve para concatenar con variables o expresiones:

```
>>> 'Py' 'thon'
'Python'
```

En el ejemplo anterior vimos lo que ocurre con dos literales, ahora veamos qué pasa si queremos utilizar este método con una variable y una cadena literal:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # no se puede concatenar una variable y una cadena literal
  ...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  ...
SyntaxError: invalid syntax
```

En este caso, para lograr la concatenacion, la solucion es usar el operador '+':

```
>>> prefix + 'thon'
'Python'
```

Una particularidad muy útil de las cadenas de texto es que se pueden indexar, el primer carácter de la cadena tiene el índice 0. No hay un tipo de dato para los caracteres; un carácter es simplemente una cadena de longitud uno:

```
>>> palabra = 'Python'
>>> palabra[0]  # caracter en la posición 0
'P'
>>> palabra[5]  # caracter en la posición 5
'n'
```

Algo que Python nos permite realizar a diferencia de otros lenguajes y que resulta de mucha utilidad es utilizar índices negativos, y de esta manera comenzar a contar de atrás para adelante en una cadena de caracteres:

```
>>> palabra[-1]  # último caracter
'n'
>>> palabra[-2]  # ante último caracter
'o'
>>> palabra[-6]
'P'
```

Es posible obtener subcadenas utilizando índices, para ello simplemente escribiremos los índices entre los cuales se contiene la subcadena que nos interesa de la siguiente manera:

```
>>> palabra[0:2]  # caracteres desde la posición 0 (incluída) hasta la 2 (excluída)
'Py'
>>> palabra[2:5]  # caracteres desde la posición 2 (incluída) hasta la 5 (excluída)
'tho'
```

El primer caracter que denotamos siempre es incluido, mientras que el último siempre se excluye. De esta forma si pusieramos s[:i]+s[i:] la concatenación nos dará la cadena s sin agregado ni omisión de caracteres:

```
>>> palabra[:2] + palabra[2:]
'Python'
>>> palabra[:4] + palabra[4:]
'Python'
```

Los índices de las subcadenas tienen valores por defecto útiles; el valor por defecto para el primer índice es cero, el valor por defecto para el segundo índice es la longitud de la subcadena a extraer.

```
>>> palabra[:2]  # caracteres desde el principio hasta la posición 2 (excluída)
'Py'
>>> palabra[4:]  # caracteres desde la posición 4 (incluída) hasta el final
'on'
>>> palabra[-2:] # caracteres desde la ante-última (incluída) hasta el final
'on'
```

Una forma de recordar cómo funcionan las extracciones de subcadenas es pensar en los índices como puntos entre caracteres, con el punto a la izquierda del primer carácter numerado en 0. Luego, el punto a la derecha del último carácter de una cadena de n caracteres tienen índice n, por ejemplo:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Como podemos observar, la primera fila de números nos da la posición de comienzo a fin con los números 0 a 6. La segunda fila en cambio nos da los índices negativos para poder llamar a cada posición desde el último hasta el primer caracter.
Para índices no negativos, la longitud de la subcadena es la diferencia de los índices, si ambos entran en los límites. Por ejemplo, la longitud de palabra[1:3] es 2.
Si no sabemos la longitud de una cadena, podemos suponer que poniendo un índice muy grande obtendremos el texto completo, pero no es así. Utilizar un índice mayor a la longitud de la cadena producirá un error:

```
>>> palabra[42]  # la palabra solo tiene 6 caracteres
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

No obstante, esto no sucede en las subcadenas:

```
>>> palabra[4:42]
'on'
>>> palabra[42:]
''
```

Otra característica de las cadenas de caracteres en Python es que no pueden ser modificadas, es decir que una vez que fueron definidas, sus elementos permanecen inmutables. Por eso, asignar a una posición indexada de la cadena resulta en un error:

```
>>> palabra[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> palabra[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

Para utilizar una cadena diferente a la creada, es necesario crear otra cadena:

```
>>> 'J' + palabra[1:]
'Jython'
>>> palabra[:2] + 'py'
'Pypy'
```

Para solucionar el problema de no conocer la longitud de la cadena de texto y poder utilizar todos sus elementos para trabajar, contamos con la función len(), la cual nos devuelve la longitud de la cadena de texto:

```
>>> s = 'supercalifrastilisticoespialidoso'
>>> len(s)
33
```

Otro tipo de dato que utilizaremos más adelante: Listas

Python tiene varios tipos de datos compuestos, usados para agrupar otros valores. El más versátil es la lista, la cual puede ser escrita como una lista de valores separados por coma (ítems) entre corchetes. Las listas pueden contener ítems de diferentes tipos, pero usualmente los ítems son del mismo tipo:

```
>>> cuadrados = [1, 4, 9, 16, 25]
>>> cuadrados
[1, 4, 9, 16, 25]
```

Continuando con las sentencias útiles

Ahora que ya sabemos cómo podemos utilizar otros tipos de datos además de los numéricos, podemos investigar un poco más sobre las sentencias que tenemos disponibles para hacer códigos interesantes en Python.

La sentencia for

En otros lenguajes de programación, hubiéramos explicado la sentencia for a la par de la sentencia while, ya que nos permitiría recorrer una serie de datos o realizar cálculos con un principio, un fin y un cierto paso. Pero en Python la sentencia for es diferente, itera sobre los ítems de cualquier secuencia (una lista o una cadena de texto), en el orden que aparecen en la secuencia. Por ejemplo:

```
>>> # Midiendo cadenas de texto
... palabras = ['gato', 'ventana', 'defenestrado']
>>> for p in palabras:
...     print(p, len(p))
...
gato 4
ventana 7
defenestrado 12
```

Si necesitáramos modificar la secuencia sobre la que se está iterando mientras estamos adentro del ciclo (por ejemplo, para borrar algunos ítems), se recomienda primero realizar una copia. Iterar sobre una secuencia no hace implícitamente una copia. La notación de subcadena es especialmente conveniente para esto:

```
>>> for p in palabras[:]:  # hace una copia por subcadena de toda la lista
...     if len(p) > 6:
...         palabras.insert(0, p)
...
>>> palabras
['defenestrado', 'ventana', 'gato', 'ventana', 'defenestrado']
```

Con for w in words:, el ejemplo intentaría crear una lista infinita, insertando defenestrado una y otra vez.

La función range()

Antes mencionamos que la sentencia for se utiliza de manera diferente en Python que en otros lenguajes, no obstante si necesitáramos iterar sobre una secuencia de números, contamos con la función integrada range(), que genera progresiones aritméticas y podríamos utilizarla en combinación con for:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

range(5) produce 5 valores comenzando en 0, es decir 0,1,2,3 y 4. El valor final no se parte de la secuencia. Es posible realizar ciertas especificaciones sobre la función range(), tales como que comience con otro valor diferente de 0 o que incremente con una cantidad particular en vez de realizar incrementos de 1, incluso es posible configurarlo en decremento:

```
range(5, 10)
  5 through 9

range(0, 10, 3)
  0, 3, 6, 9

range(-10, -100, -30)
 -10, -40, -70
```

Para iterar sobre los índices de una secuencia, es posible utilizar range() y len() en combinación:

```
>>> a = ['Mary', 'tenia', 'un', 'corderito']
>>> for i in range(len(a)):
...         print(i, a[i])
...
0 Mary
1 tenia
2 un
3 corderito
```

Si queremos realizar un print de un range para mostrar los resultados por pantalla, nos muestra algo curioso:

```
>>> print(range(10))
range(0, 10)
```

No debemos confundir el objeto que nos devuelve range() con una lista, si bien se ve de forma similar, en realidad es un objeto que devuelve los items sucesivos de la secuencia deseada pero no se construye la lista y de esa manera se ahorra espacio.

Entonces, es un objeto iterable; esto es, que se lo puede usar en funciones y construcciones que esperan algo de lo cual obtener ítems sucesivos hasta que se termine. Si queremos construir una lista, contamos con la función list() la cual creará listas a partir de objetos iterables:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Las sentencias break, continue, y else en lazos

La sentencia break, como en C, termina el lazo for o while más anidado.
Las sentencias de lazo pueden tener una cláusula else que es ejecutada cuando el lazo termina, luego de agotar la lista (con for) o cuando la condición se hace falsa (con while), pero no cuando el lazo es terminado con la sentencia break. Se ejemplifica en el siguiente lazo, que busca números primos:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'es igual a', x, '*', n/x)
...             break
...     else:
...         # sigue el bucle sin encontrar un factor
...         print(n, 'es un numero primo')
...
2 es un numero primo
3 es un numero primo
4 es igual a 2 * 2
5 es un numero primo
6 es igual a 2 * 3
7 es un numero primo
8 es igual a 2 * 4
9 es igual a 3 * 3
```

Observemos el detalle de que el else no pertenece al if, sino al ciclo for. Esto podemos observarlo fácilmente por la indentación.
Cuando se usa con un ciclo, el else tiene más en común con el else de una declaración try que con el de un if: el else de un try se ejecuta cuando no se genera ninguna excepción, y el else de un ciclo se ejecuta cuando no hay ningún break.
La declaración continue, también tomada de C, continua con la siguiente iteración del ciclo:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Encontré un número par", num)
...         continue
...     print("Encontré un número", num)
Encontré un número par 2
Encontré un número 3
Encontré un número par 4
Encontré un número 5
Encontré un número par 6
Encontré un número 7
Encontré un número par 8
Encontré un número 9
```

Errores y Excepciones en el código

Cuando comenzamos a escribir código somos más propensos a cometer algunos errores de sintaxis, y considerando que cometer errores es muy común en programación aún en los programadores más experimentados, vale mencionar los dos tipos más frecuentes de errores que nos encontraremos en la programación.
Podemos dividir los errores en *errores de sintaxis* y *excepciones*.
Podemos identificar los errores de sintaxis por haber cometido algún error al escribir una estructura, omitir algún signo de puntuación, etc.
Las excepciones en cambio son errores que se producen en tiempo de ejecución, por ejemplo, si en mi código intento acceder a un archivo de texto y el mismo no existe en la ubicación especificada, se producirá una excepción.

La sentencia pass

La sentencia pass no hace nada. Se puede usar cuando una sentencia es requerida por la sintaxis pero el programa no requiere ninguna acción. Por ejemplo:

```
>>> while True:
...     pass  # Espera ocupada hasta una interrupción de teclado (Ctrl+C)
...
```

Se usa normalmente para crear clases en su mínima expresión:

```
>>> class MyEmptyClass:
...     pass
...
```

Otro lugar donde se puede usar pass es como una marca de lugar para una función o un cuerpo condicional cuando estamos trabajando en código nuevo, lo cual te nos permite testear el código sin tener que implementar esa parte de código antes de hacer el testeo. El pass se ignora silenciosamente:

```
>>> def initlog(*args):
...     pass   # Acordate de implementar esto!
...
```

Lo que aprendimos en esta clase

A lo largo de esta clase, aprendimos a manejar diferentes tipos de datos y las sentencias básicas para comenzar a desarrollar nuestros propios códigos en Python.

Aprendimos a trabajar con números, cadenas de caracteres y listas, así como a utilizar las principales estructuras de control tales como if/else, while, for entre otras.