PREPROCESAMIENTO DE DATOS CON PYTHON

Abraham Zamudio

VI Programa de Especialización en Machine Learning con Python

2020

CTIC-UNI
BUSINESS SCHOOL

Introducción : El poder de python en Ciencias e Ingeniería

Ecosistema

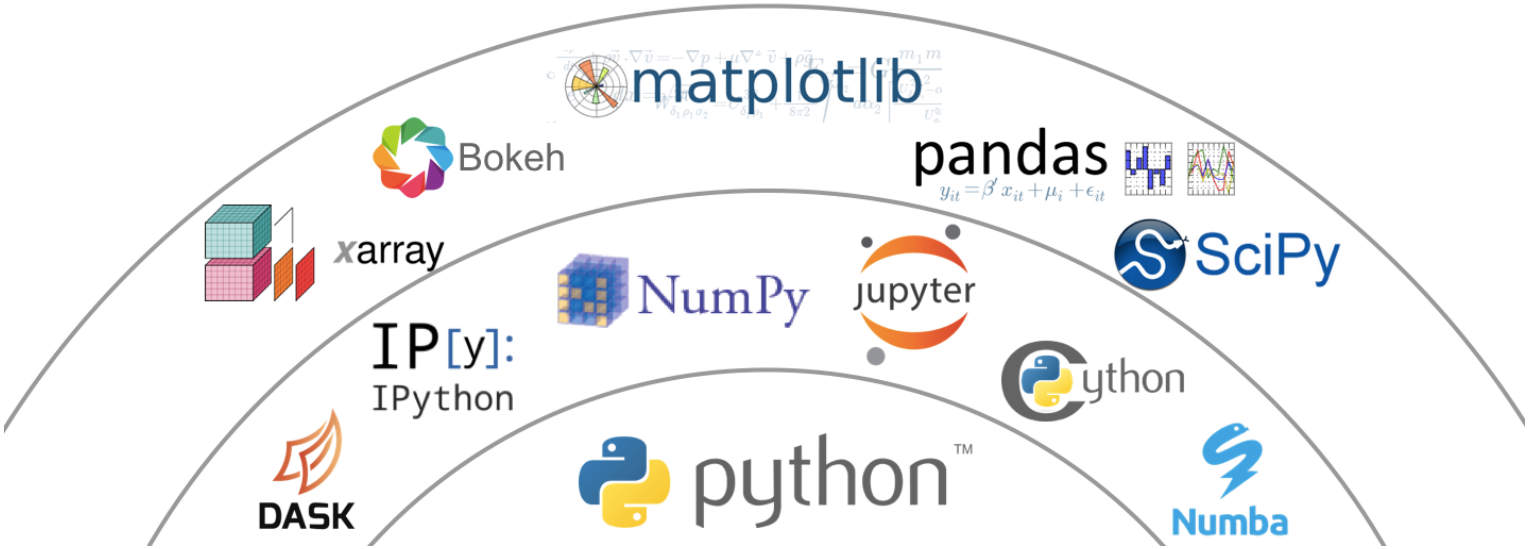


Figure: Python en Ciencias e Ingeniería

Necesidad

1. Obtener datos (simulación, control de experimentos)
2. Manipular y procesar la data.
3. Visualizar resultados, de comprensión rápida, pero también con gráficos de alta calidad , para informes o publicaciones.

Fortalezas de python

- ⋆ Baterías incluidas : Gran colección de bloques (ladrillos) ya existentes de métodos numéricos clásicos, herramientas para gráficar o procesar datos. **No** necesitaras programar el trazado de una curva o función, o como calcular una transformada de fourier o algún algoritmo de ajuste. ***No reinventes la rueda***.

Fortalezas de python

- ⋆ Fácil de aprender : A la mayoría de los profesionales no se les paga como programadores, ni se les a entrenado para eso. Necesitan poder trazar el gráfico de una función, suavizar una señal, hacer una transformada de fourier en unos pocos minutos.

Fortalezas de python

- ⋆ Comunicación fácil : Para mantener el código vivo dentro de una laboratorio de investigación o de una compañía, debe ser tan legible como un libro por parte de los que colaboradores, estudiantes o quizás clientes. La sintaxis de python es simple, evitando símbolos extraños o largas especificaciones de rutina que desviarían al lector de la comprensión matemática o científica del código.

Fortalezas de python

- ⋆ Código eficiente : Los módulos numéricos de python son computacionalmente eficientes. Pero no esta demás decir que un código rápido se vuelve inútil si se dedica demasiado tiempo en escribirlo. Python apunta a tiempos de desarrollo rápidos y tiempos de ejecución rápido.

Fortalezas de python

- ⋆ Universal : Python es un lenguaje usado para problemas diferentes. Aprender python evita aprender un nuevo software cada nuevo problema.

Introducción : El poder de python en Ciencias e Ingeniería

¿ Como se compara python con otras soluciones ?

Lenguajes compilados : C,C++, Fortran ...

- ⋆ **Pros** :Muy rápido. Para cálculos pesados, es difícil superar a estos idiomas.
- ⋆ **Contras** :Uso doloroso: sin interactividad durante el desarrollo, pasos de compilación obligatorios, sintaxis detallada, administración de memoria manual. Estos son lenguajes difíciles para no programadores.

Introducción : El poder de python en Ciencias e Ingeniería

¿ Como se compara python con otras soluciones ?

Matlab

- ⋆ **Pros** : Muy rica colección de bibliotecas con numerosos algoritmos implementados, para muchos dominios de aplicaciones diferentes. Ejecución rápida porque estas bibliotecas a menudo se escriben en un lenguaje compilado.Entorno de desarrollo agradable: completo y con manual de ayuda muy documentado, editor integrado, etc. El soporte comercial está disponible.
- ⋆ **Contras** : El idioma base es bastante deficiente y puede volverse restrictivo para usuarios avanzados.No es gratis.

Introducción : El poder de python en Ciencias e Ingeniería

¿ Como se compara python con otras soluciones ?

Julia

- ⋆ **Pros** : Código rápido, pero interactivo y simple. Se conecta fácilmente a Python o C.
- ⋆ **Contras** : Ecosistema limitado a computación numérica. Aún joven.

Introducción : El poder de python en Ciencias e Ingeniería

¿ Como se compara python con otras soluciones ?

Otros lenguajes scripting : Scilab, Octave, IDL , etc

- ⋆ **Pros** : De código abierto, gratuito, o al menos más barato que Matlab.Algunas características pueden ser muy avanzadas (estadísticas en R, imágenes satelitales en IDL , etc.)
- ⋆ **Contras** : Hay menos algoritmos disponibles que en Matlab, y el lenguaje no es más avanzado.Algunos programas están dedicados a un dominio. Ej: Gnuplot para dibujar curvas. Estos programas son muy potentes, pero están restringidos a un solo tipo de uso, como el trazado de curvas.

Introducción : El poder de python en Ciencias e Ingeniería

¿ Como se compara python con otras soluciones ?

Python

- ⋆ **Pros** : Muy buenas y bien documentadas bibliotecas para computación científica.Lenguaje bien pensado, que permite escribir código muy legible y bien estructurado: `codificamos lo que pensamos`.Muchos módulos más allá de la computación científica (servidor web, acceso a puerto serie, etc.).Software libre y de código abierto, ampliamente difundido, con una comunidad vibrante.Una variedad de entornos potentes para trabajar, como `IPython, Spyder, Jupyter notebooks,Pycharm`.
- ⋆ **Contras** : No todos los algoritmos se pueden encontrar en módulos.

Empecemos con formulas

¿Porque?

1. Problemas fáciles de entender.
2. Muchos conceptos fundamentales son introducidos
   - Variables.
   - Expresiones Aritméticas.
   - Impresión de textos y números.

Evaluemos una formula matemática : MRUV

$$y(t) = v_0\, t - \frac{1}{2}\, g\, t^2$$

donde

1. y es la altura (posición) como función del tiempo.
2. v_0 es la velocidad inicial en el tiempo cero.
3. g es la aceleración de la gravedad.

Tarea : Dados v_0, g, t, calcular y.

¿Usar una calculadora? Un programa es mucho mas poderoso

¿Que es un programa?

Una secuencia de instrucciones para la computadora, escrita en un lenguaje de programación, algo así como el inglés, pero mucho más simple, y mucho más estricto.

La sintaxis tipo *printf* (C/C++) da una gran flexibilidad en el formato de texto con números

La salida de nuestros cálculos a menudo contienen texto y números, por ejemplo para $t = 0.6\,s, y = 1.23\,m$

Función print

Queremos controlar el formateo de los números, no de decimales :

Estilo : 0.6 vs. $6E-01$ o $6.0e-0.1$.

Aquí es donde el formateo que obtenemos con `print` es muy útil para este propósito:

MRUV : Version1

```
v0 = 5
g = 9.81
t = 0.6
y = v0*t - 0.5*g*t**2
print(y)
```

Nombre de las Variables

Como elegir los nombres de las variables :

1. Use los mismos nombres para la variable en el programa que en la descripción matemática del modelo (problema) que desea resolver.
2. Para todas las variables que no tengan una definición matemática precisa, use un nombre descriptivo cuidadosamente seleccionado.

MRUV: Version2

```
initial_velocity = 5
acceleration_of_gravity = 9.81
TIME = 0.6
VerticalPositionOfBall = initial_velocity*TIME - \
      0.5*acceleration_of_gravity*TIME**2
print(VerticalPositionOfBall)
```

Ayuda de Python

```
1 help()
```

```
Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.6/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".


help>
```

Palabras reservadas

Después de ejecutar : help()

```
help> keywords
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

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

Opciones de la función print (MRUV : Version3)

```
1 initial_velocity = 5
2 acceleration_of_gravity = 9.81
3 TIME = 0.6
4 VerticalPositionOfBall = initial_velocity*TIME - \
5    0.5*acceleration_of_gravity*TIME**2
6 print("At t=%g s, the height of the ball is %.2f m." % \
7    (TIME,VerticalPositionOfBall))
```

Formateando texto y números

Aqui podemos ver una lista con las mas importantes opciones de print

Format	Meaning
`%s`	a string
`%d`	an integer
`%0xd`	an integer in a field of with x, padded with leading zeros
`%f`	decimal notation with six decimals
`%e`	compact scientific notation, `e` in the exponent
`%E`	compact scientific notation, `E` in the exponent
`%g`	compact decimal or scientific notation (with `e`)
`%G`	compact decimal or scientific notation (with `E`)
`%xz`	format `z` right-adjusted in a field of width `x`
`%-xz`	format `z` left-adjusted in a field of width `x`
`%.yz`	format `z` with `y` decimals
`%x.yz`	format `z` with y decimals in a field of width `x`
`%%`	the percentage sign `%` itself

Ejemplo 1 (MRUV : Version4)

```
v0 = 5
g = 9.81
t = 0.6
y = v0*t - 0.5*g*t**2
print('En t={t:g} s, la altura de la bola es {y:.2f} m.' \
    .format(t=t, y=y)
)
```

Ejemplo 1 (MRUV : Version5)

```
1  v0 = 5
2  g = 9.81
3  t = 0.6
4  y = v0*t - 0.5*g*t**2
5
6  print( """
7  En t=%f s, una bola con
8  velocidad inicial v0=%.3E m/s
9  esta en la altura %.2f m.
10 """ % (t, v0, y)
11 )
```

Evaluemos otra formula matemática : Conversión de grados Celsius-Fahrenheit

Nuestro próximo ejemplo incluye la fórmula para convertir la temperatura medida en grados Celsius al valor correspondiente en grados Fahrenheit:

$$F = \frac{9}{5}C + 32$$

En esta fórmula, C es la cantidad de grados en grados Celsius, y F es la temperatura correspondiente medida en Fahrenheit. Nuestro objetivo ahora es escribir un programa de computadora que pueda calcular F cuando se conoce a C.

Python 2.x vs. Python 3.y

```
abraham@DarkMath666:~$ python2
Python 2.7.12 (default, Dec  4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> C = 21
>>> F = (9/5)*C + 32
>>> print(F)
53
>>>
```

```
abraham@DarkMath666:~$ python
Python 3.6.4 |Anaconda custom (64-bit)| (default, Jan 16 2018, 18:10:19)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> C = 21
>>> F = (9/5)*C + 32
>>> print(F)
69.80000000000001
>>>
```

Algunos enlaces :

- ▸ python 2.x vs. python 3.x
- ▸ Diferencias py2.7 vs. py3.x

Python 2.x vs. Python 3.y

```
abraham@DarkMath666:~$ python2
Python 2.7.12 (default, Dec  4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> C = 21
>>> F = (9/5)*C + 32
>>> print(F)
53
>>> type(F)
<type 'int'>
>>>
```

```
abraham@DarkMath666:~$ python
Python 3.6.4 |Anaconda custom (64-bit)| (default, Jan 16 2018, 18:10:19)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> C = 21
>>> F = (9/5)*C + 32
>>> print(F)
69.80000000000001
>>> type(F)
<class 'float'>
>>>
```

Importando módulos : Evaluando funciones matemáticas estándar

Las fórmulas matemáticas frecuentemente involucran funciones como *sin*, *cos*, *tan*, *sinh*, *cosh*, *exp*, *log*, etc. En una calculadora de bolsillo, usted tiene botones especiales para tales funciones. Del mismo modo, en un programa, escrito en python, también tiene una funcionalidad ya preparada para evaluar estos tipos de funciones matemáticas. Uno podría, en principio, escribir su propio programa para evaluar, por ejemplo, la función $sin(x)$, pero cómo hacerlo de manera eficiente es un tema no trivial. Los expertos han trabajado en este problema durante décadas e implementado sus mejores recetas en piezas de software que deberíamos reutilizar.

Importando módulos : Evaluando funciones matemáticas estándar

Para aquellos que no conocen las matemáticas, es difícil sentir la belleza, la profunda belleza de la naturaleza... Si quieres aprender sobre la naturaleza, apreciar la naturaleza, es necesario aprender el lenguaje en el que habla.

(Richard Feynman)

Importando módulos : Evaluando funciones matemáticas estándar

Problema Considere la fórmula para calcular la altura de una bola en movimiento vertical, con una velocidad inicial hacia arriba v_0:

$$y = v_o\, t - \frac{1}{2}\, g\, t^2$$

$$0 = \frac{1}{2}\, g\, t^2 - v_0\, t + y$$

Entonces :

$$t_1 = \frac{v_0 - \sqrt{v_0^2 - 2gy}}{g}$$

$$t_2 = \frac{v_0 + \sqrt{v_0^2 - 2gy}}{g}$$

Importando módulos : Evaluando funciones matemáticas estándar

El programa para evaluar las expresiones para t_1 y t_2 en un programa de computadora, necesitamos acceso a la función de raíz cuadrada. En Python, la función de raíz cuadrada y muchas otras funciones matemáticas, como *sin*, *cos*, *sinh*, *exp*, y *log*, están disponibles en un módulo llamado `math`. Primero debemos importar el módulo antes de que podamos usarlo, es decir, debemos escribir `import math`. A partir de entonces, para tomar la raíz cuadrada de una variable *a*, podemos escribir `math.sqrt(a)`. Esto se demuestra en un programa para calcular t_1 y t_2:

Importando módulos : Evaluando funciones matemáticas estándar

Modulo *math*

```
1 v0 = 5
2 g = 9.81
3 y = 0.2
4 import math
5 t1 = (v0 - math.sqrt(v0**2 - 2*g*y))/g
6 t2 = (v0 + math.sqrt(v0**2 - 2*g*y))/g
7 print('En t=%g s y %g s, la altura es %g m.' %(t1, t2, y))
```

Dos formas de importar un módulos

Primer camino

```
1 import math
2 # y acceder a las funciones de manera individual
3 x = math.sqrt(y)
```

Segundo camino

```
1 from math import sqrt
2 # Ahora podemos trabajar con sqrt de manera directa
3 # , sin la necesidad de usar el prefijo math
4 # De esta manera se puede importar mas de una funcion
5 # del modulo math
6 from math import sqrt, exp, log, sin
```

Dos formas de importar un módulos

Consulta la ayuda del modulo math : help()

```
1 help> math
```

```
Help on module math:

    math


    https://docs.python.org/3.6/library/math

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.


    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.


        (...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

         (...)
:
```

Otra version de la segunda forma

Importando todos las funciones del modulo math

```
1 from math import *
```

Esta ultima linea sirve para importar todas las funciones del modulo `math`.Importar todas las funciones de un modulo usando el carácter * es conveniente, pero da lugar a una gran cantidad de palabras reservadas que no se pueden usar. **En general** se recomienda solo importar las funciones que se van a usar.

Importando con nuevos nombres

Los módulos y funciones importados pueden recibir nuevos nombres en la sentencia `import`.

Importando todos las funciones del modulo math

```
1 import math as m
2 # m es ahora el nombre del modulo math
3 v = m.sin(m.pi)
4 from math import log as ln
5 v = ln(5)
6 from math import sin as s, cos as c, log as ln
7 v = s(x)*c(x) + ln(x)
```

Ejercicio 1: Calcular el sinh(x)

Nuestro primer ejercicio implica hacer llamadas a algunas funciones matemáticas del modulo `math`.Veamos la definición de la función $sinh(x)$

$$sinh(x) = \frac{1}{2}(e^x - e^{-x})$$

Hacer el calculo de tres formas diferentes :

- Llamando a la función `math.sinh`.
- Calculando el lado derecho usando `math.exp`.
- Calculando el lado derecho usando el operador exponencial (**) : `math.e**x`

Ejercicio 2: Distancia entre dos puntos

Como segundo ejercicio también haremos llamadas a algunas funciones matemáticas del modulo `math`.Vamos a calcular la distancia entre dos puntos del plano cartesiano ($\mathbb{R}^2$). Para ello, sean $A = (x_1, y_1)$ y $B = (x_2, y_2)$, entonces la distancia entre A y B se calcula haciendo uso del *Teorema de Pitagoras*.

$$d(A, B) = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$$

Hacer el calculo para $A = (-1, 2)$ y $B = (3, 6)$.

Ejercicio 3: Listado de funciones del modulo math

```
1 import math
2 math_ls = dir(math)
3 print(math_ls)
```

Usa como mínimo 5 funciones que no conozcas de las funciones listadas en la salida de la linea 3, para construir un par de ejemplos de calculo.

Ejercicio 4: Numeros complejos en python

```
a = -2
b = 0.5
# Creamos numeros complejos con las variables
s = complex(a, b)
print(s)
w= s.conjugate()
s*w
s/w
s.real
s.imag
```

Lea lo concerniente al modulo *cmath*.

▸ Funciones matematicas para numeros complejos.

Computación simbólica con python : Sympy

Wikipedia

En matemáticas y ciencias de la computación, el álgebra computacional, también conocida como cálculo simbólico o cálculo algebraico, es un área científica que se refiere al estudio y desarrollo de algoritmos y software para la manipulación de expresiones matemáticas y otros objetos matemáticos. Aunque, hablando con propiedad, el álgebra computacional debe ser un sub-campo de la computación científica, ellos son considerados generalmente como campos distintos, porque la computación científica se basa generalmente en el análisis numérico con números aproximados en punto flotante; mientras que, el álgebra computacional enfatiza el cálculo exacto con expresiones que contengan variables que no tienen cualquier valor dado y por lo tanto son manipulados como símbolos (de ahí se debe el nombre de cálculo simbólico).

Computación simbólica con python : Sympy

Sistema de álgebra computacional

Un sistema algebraico computacional o sistema de álgebra computacional (CAS, del inglés computer algebra system) es un programa de ordenador o calculadora avanzada que facilita el cálculo simbólico. La principal diferencia entre un CAS y una calculadora tradicional es la habilidad del primero para trabajar con ecuaciones y fórmulas simbólicamente, en lugar de numéricamente. Es decir, una expresión como a + b es interpretada siempre como "la suma de dos variables", y no como "la suma de dos números" (con valores asignados).

Computación simbólica con python : Sympy

Sistema de álgebra computacional : Caracteristicas (I)

- ⋆ Simplificación de una expresión a la forma más simple o a una forma estándar.
- ⋆ Cambio en la forma de las expresiones: expansión de productos y potencias, factorización, reescritura de un cociente de polinomios como suma de fracciones parciales, reescritura de funciones trigonométricas como exponenciales (y viceversa), etc.
- ⋆ Operaciones con matrices incluyendo productos de matrices, inversa de una matriz, etc.
- ⋆ Resolución de algunas ecuaciones.
- ⋆ Cálculo de algunos límites de funciones.
- ⋆ Cálculo de derivadas y derivadas parciales.
- ⋆ Cálculo de algunas integrales indefinidas, definidas y de algunas transformadas integrales.

Computación simbólica con python : Sympy

Sistema de álgebra computacional : Caracteristicas (II)

- ⋆ Aproximación de funciones por desarrollo en series de potencias.
- ⋆ Resolución de algunas ecuaciones diferenciales.
- ⋆ Manipulación exacta de fracciones y radicales.
- ⋆ Realización de operaciones con precisión arbitraria.
- ⋆ Respecto a la presentación de resultados: Visualizado de las expresiones matemáticas en una forma bidimensional, usando con frecuencia sistemas de composición similares a LaTeX.

Algunos de los CAS más destacados son:

- Mathematica
- Maple
- SageMath
- Matlab
- Magma

Computación simbólica con python : Sympy

Primeros pasos

```
1  from sympy import *
2
3  # cuatro variables simbolicas son creadas, los valores
4  # previos de estas variables son sobreescritos.
5  x, y, a, b = symbols ('x y a b')
6
7  # La variable f se convierte automáticamente
8  # en un caracter.
9  f = a**3 * x + 3 * a**2 * x**2/2 + a * x**3 + x**4/4
10 type (f)
11
12 var ('u, v')
13 # La variable f se convierte automáticamente
14 # en un caracter
15 f = sin (u) ** 2 + tan (v)
16 type (f)
```

La principal diferencia entre las funciones de symbols() y var() es que la primera función devuelve una referencia a un objeto de tipo carácter (character object). Para su uso en el futuro, se le debe asignar una variable. El segundo, sin asignación, crea una variable de carácter.

```
1 import sympy as sp
2 a = sp.Rational(1,2)
3
4 ARacional = sp.Rational(2)**50/sp.Rational(10)**50
5 Afloat = 2**50/10**50
6
7 sp.pi**2
8 sp.pi.evalf()
9 (sp.pi + sp.exp(1)).evalf()
```

Para hacer operaciones simbólicas hay que definir explícitamente los símbolos que vamos a usar, que serán en general las variables y otros elementos de nuestras ecuaciones:

```
x = sp.Symbol('x')
y = sp.Symbol('y')
# Podemos manipular los símbolos como queramos
x+y+x-y
(x+y)**2
((x+y)**2).expand()
# Es posible hacer una sustitución usando la función subs
((x+y)**2).subs(x, 1)
((x+y)**2).subs(x, y)
```

Computación simbólica con python : Sympy

Primeros pasos : Calculo con limites

$$\left[\lim_{x \to c} f(x) = L\right] \leftrightarrow \left[\forall \epsilon > 0 \exists \delta > 0 : 0 < |x - c| < \delta \rightarrow |f(x) - L| < \epsilon\right]$$

```
1 x = sp.Symbol("x")
2 sp.limit(sin(x)/x, x, 0)
3
4 # Limites infinitos
5 sp.limit(x, x, oo)
6 sp.limit(1/x, x, oo)
7 sp.limit(x**x, x, 0)
```

Computación simbólica con python : Sympy

Primeros pasos : Cálculo de derivadas

```
x = sp.Symbol('x')
diff(sp.sin(x), x)
diff(sp.sin(2*x), x)
diff(sp.tan(x), x)

# este ultimo resultado se puede comprobar :
dx = sp.Symbol('dx')
sp.limit( (tan(x+dx)-tan(x) )/dx, dx, 0)
```

Computación simbólica con python : Sympy

Primeros pasos : Cálculo de derivadas de orden superior

```
# Derivada de orden 1
sp.diff(sin(2*x), x, 1)

# Derivada de orden 2
sp.diff(sin(2*x), x, 2)

# Derivada de orden 3
sp.diff(sin(2*x), x, 3)
```

Para la expansión de series se aplica el método `series(var, punto, orden)` a la función que se desea expandir:

```
cos(x).series(x, 0, 10)
(1/cos(x)).series(x, 0, 10)

e = 1/(x + y)
s = e.series(x, 0, 5)

# La función pprint de Sympy imprime el resultado
# de manera más legible:
pprint(s)
```

Computación simbólica con python : Sympy

Primeros pasos : Integración simbólica (I)

```
sp.integrate(6*x**5, x)
sp.integrate(sp.sin(x), x)
sp.integrate(sp.log(x), x)
sp.integrate(2*x + sp.sinh(x), x)
sp.integrate(exp(-x**2)*erf(x), x)

# También es posible calcular integrales definidas:
sp.integrate(x**3, (x, -1, 1))
sp.integrate(sin(x), (x, 0, pi/2))
sp.integrate(cos(x), (x, -pi/2, pi/2))
```

Computación simbólica con python : Sympy

Primeros pasos : Integración simbólica (II)

```
# Y también integrales impropias:
sp.integrate(exp(-x), (x, 0, oo))
sp.integrate(log(x), (x, 0, 1))

# Algunas integrales definidas complejas es necesario
# definirlas como objeto Integral() y luego evaluarlas
# con el método evalf():
integ = sp.Integral(sin(x)**2/x**2, (x, 0, oo))
integ.evalf()
```

SymPy es capaz de resolver ecuaciones algebraicas de una o varias variables:

```
# f(x)=0
solve(x**4 - 1, x)

# Es capar de resolver múltiples ecuaciones respecto a
# múltiples variables (sistemas de ecuaciones)
# proporcionando una tupla como segundo argumento:
solve([x + 5*y - 2, -3*x + 6*y - 15], [x, y])

# También tiene capacidad (limitada) de resolver
# ecuaciones transcendentales:
solve(exp(x) + 1, x)
```

Otra alternativa, en el caso de ecuaciones polinómicas, es factor. factor devuelve el polinomio factorizado en términos irreducibles y es capaz de calcular la factorización sobre varios dominios:

```
1 f = x**4 - 3*x**2 + 1
2 factor(f)
```

Computación simbólica con python : Sympy

Ejercicio 5

- Calcular $\sqrt{2}$ con 100 decimales
- Calcular $\frac{1}{2} + \frac{1}{3}$ en aritmética racional.
- Calcular la forma expandida de $(x + y)^6$.
- Calcular $\lim_{x \to 0} \frac{\mathrm{sen(x)}}{x}$
- Resuelve el sistema de ecuaciones :

$$\begin{cases} x + y = 2 \\ 2x + y = 0 \end{cases}$$

Entrada de Usuario (input)

Consideremos el programa para evaluar la formula

$$x = A \sin(\omega\, t)$$

Ecuación de una Onda

```
1 from math import sin
2 A = 0.1
3 w = 1
4 t = 0.6
5 x = A*sin(w*t)
6 print(x)
```

En este programa, A, w y t son datos de entrada en el sentido de que estos parámetros deben ser conocidos antes de que el programa pueda realizar el cálculo de x. Los resultados producidos por el programa, en este caso la variable x, constituyen los datos de salida.

Entrada de Usuario (input)

Entrada por teclado

Conversión de grados Celsius-Fahrenheit

```
1 C = input('C=? ')
2 C = float(C)
3 F = 9.0/5*C + 32
4 print(F)
```

Saludo

```
1 name = str(input("Cual es tu nombre? "))
2 print("Mucho gusto " + name + "!")
3 age = input(" Cual es tu edad ? ")
4 print("Ya tienes %s anios de edad  %s !" % (age,name))
```

Entrada de Usuario (input)

Entrada por linea de comandos

Conversión de grados Celsius-Fahrenheit (c2f_cli.py)

```
1 import sys
2 C = float(sys.argv[1])
3 F = 9.0/5*C + 32
4 print(F)
```

Ejecutar

```
1 %run c2f_cli.py 21
```

Entrada de Usuario (input)

Entrada por linea de comandos

MRUV

```
1 import sys
2 t = float(sys.argv[1])
3 v0 = float(sys.argv[2])
4 g = 9.81
5 y = v0*t - 0.5*g*t**2
6 print(y)
```

Ejecutar

```
1 %run mruv_cli.py 0.6 5
```

Bucles y listas

Crea una table de grados celsius y fahrenheit

```
-20  -4.0
-15   5.0
-10  14.0
-5   23.0
0    32.0
5    41.0
10   50.0
15   59.0
20   68.0
25   77.0
30   86.0
35   95.0
40 104.0
```

¿Como seria un programa que escriba la tabla anterior?

Creando la tabla : Una versión simple

Conocemos como calcular linea por linea

```
1 C = -20
2 F = 9.0/5*C + 32
3 print(C, F)
```

Seria suficiente repetir estas sentencias

```
C = -20; F = 9.0/5*C + 32 ;print(C, F)
C = -15; F = 9.0/5*C + 32 ;print(C, F)
...
C = 35; F = 9.0/5*C + 32 ;print(C, F)
C = 40; F = 9.0/5*C + 32 ;print(C, F)
```

Creando una tabla : Características de la primera versión

- Muy aburrido de escribir y fácil de introducir errores de impresión.
- Cuando la programación se vuelve aburrida, usualmente hay una **construcción** que automatiza la escritura
- La computadora es extremadamente buena para realizar tareas repetitivas.
- Para este proposito usamos bucles (loops).

Sentencia While

El bucle while hace posible repetir tareas similares.Un bucle while ejecuta repetidamente un conjunto de sentencias siempre que una condición booleana sea verdadera.

```
while condicion:
    <sentencia 1>
    <sentencia 2>
    ...
<primera sentencia despues del bucle>
```

- Todas las sentencias dentro del bucle deben estar indentadas.
- El ciclo finaliza cuando se encuentra una declaración que no este indentada.

El bucle while para crear la tabla

```
1  print ('-------------------')# cabecera de la tabla
2  C = -20                     # inicia los valores de C
3  dC = 5                      # define el incremento
4  while C <= 40:              # inicia el bucle y la condicion
5      # primera sentencia dentro del bucle
6      F = (9.0/5)*C + 32
7      # segunda sentencia dentro del bucle
8      print(C, F)
9      # ultima sentencia dentro del bucle
10     C = C + dC
11 print ('-------------------')# fin de la tabla
```

Flujo del programa dentro del bucle while

```
1 C = -20
2 dC = 5
3 while C <= 40:
4     F = (9.0/5)*C + 32
5     print(C, F)
6     C = C + dC
```

▸ Visualizacion de le ejecucion

Simulemos el ciclo while a mano

- Primero $C = -20$, como $-20 \leq 40$ tiene valor booleano verdadero, entonces ejecutamos las sentencias que están dentro del bucle.
- Calculamos F, imprimirmos y actualizamos C a -15.
- Saltamos para nuevamente evalular la condicion, evaluamos $C \leq 40$ lo cual es verdadero, luego nuevamente ejecutamos las sentencias del bucle.
- Continuamos hasta que C tome el valor de 45.
- Ahora la condicion del bucle $45 \leq 40$ tiene valor booleano FALSO, y el programa salta a la primera linea despues del bucle while.

While : Ejemplo (1) de la flexibilidad en la variable de control

```
1 i = 1
2 while i <= 50:
3     print(i)
4     i = 3*i + 1
5 print("Programa terminado")
```

While: Ejemplo (2) para verificar una condición sobre el dato de entrada

```
1 numero = int(input("Escriba un número positivo: "))
2 while numero < 0:
3     print("¡Ha escrito un número negativo! Inténtelo de nuevo")
4     numero = int(input("Escriba un número positivo: "))
5 print("Gracias por su colaboración")
```

While: Ejemplo (3) para verificar una condición sobre el dato de entrada

```
promedio, total, contar = 0.0, 0, 0

print("Introduzca la nota de un estudiante (-1 para salir): ")
nota = int(input("Ingrese la nota"))
while nota != -1:
    total = total + nota
    contar = contar + 1
    print("Introduzca la nota de un estudiante (-1 para salir):")
    nota = int(input("Ingrese la nota"))
promedio = total / contar
print("Promedio de notas del grado escolar es: " +
↪  str(promedio))
```

Expresiones Booleanas : V ó F

Una expresión con valores verdadero (true) o falsa (false) es llamada expresión booleana. Ejemplos :

```
IPython 7.0.1 -- An enhanced Interactive Python.

In [1]: C=41

In [2]: C!=40
Out[2]: True

In [3]: C<40
Out[3]: False

In [4]: C>40
Out[4]: True
```

Combinando expresiones booleanas

Muchas condiciones pueden ser combinaciones de `and` y `or` :

```
while condition1 and condition2:
    ...
while condition1 or condition2:
    ...
```

Reglas de la lógica proposicional

Regla 1 : $C1$ y $C2$ es verdadero si ambas son verdaderas.

Regla 2 : $C1$ o $C2$ es verdadero si una es verdadera.

Ejemplos de expresiones booleanas

```
In [6]: x = 0; y = 1.2

In [7]: x >= 0 and y < 1
Out[7]: False

In [8]: x >= 0 or y < 1
Out[8]: True

In [9]: x > 0 or y > 1
Out[9]: True

In [10]: x > 0 or not y > 1
Out[10]: False

In [11]: -1 < x <= 0
Out[11]: True

In [12]: not (x > 0 or y > 0)
Out[12]: False
```

Las listas son objetos para almacenar una secuencia de cosas (objetos)

Hasta ahora una variable se ha referido a un numero (o a una cadena), pero algunas veces resulta natural tener una colección de números, por ejemplo la tabla de grados celsius y fahrenheit : -20,-15,-10,-5,0,...,40. Una solucion simple pero poco eficiente es asignar una variable a cada valor

```
C1 = -20
C2 = -15
C3 = -10
...
C13 = 40
```

Las listas son objetos para almacenar una secuencia de cosas (objetos)

Pero es tonto y aburrido si es que tenemos muchos valores.Lo mejor es tener un conjunto de valores recolectados en una lista.

```
1 C = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
```

Definicion de Listas

Una lista es una estructura de datos en Python que es una secuencia de elementos ordenada y modificable (mutable). Cada elemento o valor que está dentro de una lista se llama un item. Al igual que las cadenas se definen como caracteres entre comillas, las listas se definen al tener valores entre corchetes [].

Operaciones con listas: inicializacion e indexado

Se inicializa con corchetes y coma entre cada uno de los objetos:

```
1 L1 = [-91, 'a string', 7.2, 0]
```

Los elementos son accesados via indices : $L1[3]$ ($indice = 3$).EL conjunto de indices empiezan en cero : `0,1,2,...,len(L1)-1`.

```
1  In [14]: mylist = [4, 6, -3.5]
2
3  In [15]: print(mylist[0])
4  4
5
6  In [16]: print(mylist[1])
7  6
8
9  In [17]: print(mylist[2])
10 -3.5
11
12 In [18]: len(mylist)
13 Out[18]: 3
```

Operaciones con listas : append,extend,insert,delete

```
In [20]: C = [-10, -5, 0, 5, 10, 15, 20, 25, 30]

In [21]: C.append(35) # agrega el elemtnso 35 al final

In [22]: C
Out[22]: [-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]

In [23]: C = C + [40, 45] # extiende C

In [24]: C
Out[24]: [-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]

In [25]: C.insert(0, -15) # inserta -15 en el indice 0

In [26]: C
Out[26]: [-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]

In [27]: del C[2] # borra el tercer elemento

In [28]: C
Out[28]: [-15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

Operaciones con listas : busqueda de elementos , indices negativos

```
In [30]: C.index(10) # indice del primer elemento con valor 10
Out[30]: 4

In [31]: 10 in C  # 10 es un elemento en C?
Out[31]: True

In [32]: C[-1] # el ultimo elemento de la lista
Out[32]: 45

In [33]: C[-2] # el penultimo elemento de la lista
Out[33]: 40

In [34]: somelist = ['book.tex', 'book.log', 'book.pdf']

In [35]: texfile, logfile, pdf = somelist # asignacion directa

In [36]: texfile
Out[36]: 'book.tex'

In [37]: logfile
Out[37]: 'book.log'
```

Operaciones con listas : Modificando listas con operadores (I)

Los operadores pueden ser utilizados para hacer modificaciones a las listas. Veamos cómo utilizar los operadores + y *.

```
1 # El operador + se puede utilizar para concatenar
2 # dos o más listas
3 sea_creatures = ['shark', 'octopus', 'blobfish', 'mantis
  ↪  shrimp', 'anemone']
4 oceans = ['Pacific', 'Atlantic', 'Indian', 'Southern', 'Arctic']
5
6 print(sea_creatures + oceans)
```

Operaciones con listas : Modificando listas con operadores (II)

El operador * se puede utilizar para multiplicar listas. Quizás necesite hacer copias de todos los archivos de un directorio en un servidor o compartir una lista de reproducción con amigos; en estos casos, deberá multiplicar las colecciones de datos.

```
1 print(sea_creatures * 2)
2 print(oceans * 3)
```

Operaciones con listas : Función enumerate()

La función **enumerate()** permite iterar sobre los indices y los elementos de una lista.

```
1 alist = ['a1', 'a2', 'a3']
2
3 for i, a in enumerate(alist):
4     print(i, a)
```

Operaciones con listas : Función zip()

La funcion **zip()** permite sobre dos listas en paralelo

```
1 alist = ['a1', 'a2', 'a3']
2 blist = ['b1', 'b2', 'b3']
3
4 for a, b in zip(alist, blist):
5     print(a, b)
```

Operaciones con listas : enumerate + zip

Aquí se explica cómo iterar sobre dos listas y sus índices usando enumerate junto con zip:

```
1 alist = ['a1', 'a2', 'a3']
2 blist = ['b1', 'b2', 'b3']
3
4 for i, [a, b] in enumerate(zip(alist, blist)):
5    print(i, a, b)
```

Bucle sobre los elementos de una lista con una estructura for

Usamos una estructura for para recorrer una lista y procesar cada elemento.

```
1 degrees = [0, 10, 20, 40, 100]
2 for C in degrees:
3     print('Grados celsius:', C)
4     F = 9/5.*C + 32
5     print('Fahrenheit:', F)
6 print ('La lista tiene', len(degrees), 'elementos')
```

▸ Visualizacion de le ejecucion

Como con la estructura while, las sentencias dentro de for estan indentadas.

Simulamos el bucle a mano

```
1 degrees = [0, 10, 20, 40, 100]
2 for C in degrees:
3     print(C)
4 print ('La lista tiene', len(degrees), 'elementos')
```

- Primer paso : $C = 0$
- segundo paso : $C = 10$ y así
- tercer paso : $C = 20$ y así
- quinto paso : $C = 100$, ahora el bucle for termina y el flujo del programa salta a la primera sentencia con la misma indentación que la sentencia for : `for C in degrees:`

Creando una tabla con el bucle for

```
1 Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15,20, 25, 30, 35, 40]
2 for C in Cdegrees:
3     F = (9.0/5)*C + 32
4     print( C, F)
```

Note que `print( C, F)` tiene una salida no muy precisa, usemos lo que ya vimos de la funcion `print` para formatear las dos columnas.

```
1 Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15,20, 25, 30, 35, 40]
2 for C in Cdegrees:
3     F = (9.0/5)*C + 32
4     #print( C, F)
5     print ('%5d %5.1f' % (C, F))
```

Un bucle for siempre se puede traducir a un bucle while

```
# bucle for
for elemento in algunaLista:
    # procesa elemento

# siempre se puede transformar en un while
indice = 0
while indice < len(algunaLista):
    elemento = algunaLista[indice]
    # procesa elemento
    indice += 1
```

Pero no todo bucle while se puede expresar como un bucle for.

Version que usa while para crear la tabla

```
1 Cdegrees = [-20, -15, -10, -5, 0, 5, 10,15, 20, 25, 30, 35, 40]
2 index = 0
3 while index < len(Cdegrees):
4     C = Cdegrees[index]
5     F = (9.0/5)*C + 32
6     print( '%5d %5.1f' % (C, F))
7     index += 1
```

Implementacion de una sumatoria via un bucle for

$$S = \sum_{i=1}^{N} i^2$$

```
1  N = 14
2
3  S = 0
4  for i in range(1, N+1):
5      S += i**2
6
7  # otra forma  (menos comun):
8  S = 0
9  i = 1
10 while i <= N:
11     S += i**2
12     i += 1
13
14 # Las sumatorias aparecen con frecuencia, asi que recuerda
15 # la implementacion.
```

Almacenando la tabla de grados celsius-fahrenheit en una lista

Pongamos los valores de los grados fahrenheit en una lista:

```
1 Cdegrees = [-20, -15, -10, -5, 0, 5, 10,15, 20, 25, 30, 35, 40]
2 Fdegrees = []  # empezamos con una lista vacia
3 for C in Cdegrees:
4     F = (9.0/5)*C + 32
5     Fdegrees.append(F) # agregamos el nuevo elemento a Fdegrees
6     print(Fdegrees)
```

▸ Visualizacion de le ejecucion

Bucle for sobre una lista de indices

```
1 for elemento in algunaLista:
2     # procesa elemento
3
4 # de manera alternativa podemos iterar sobre
5 # los indices de la lista
6
7 for i in range(0, len(algunaLista), 1):
8     element = algunaLista[i]
9     # procesa elemento (algunaLista[i]) directamente
```

help(range)

¿Como podemos cambiar los elementos de una lista?

Digamos que queremos sumar 2 a todos los elementos de una lista

```
1 v = [-1, 1, 10]
2 for e in v:
3     e = e + 2
4 v # [-1, 1, 10] no a sufrido cambios
```

Cambiar un elemento de lista requiere asignación de índices

Dentro del bucle, *e* es un ordinario entero,la primera vez toma el valor de -1,la siguiente vez toma el valor de 1 y termina tomando el valor de 10, sin embargo la lista (v) permanece sin cambios.
Solucion : Hay que indexar los elementos de la lista para cambiar sus valores.

```
1 v = [-1, 1, 10]
2 for i in range(len(v)):
3     v[i] = v[i] + 2
4 v
```

Enumeracion de Listas : Creacion compacta de listas

Ejemplo : Calculo de dos listas en un bucle for

```
1 n = 16
2 Cdegrees = [];Fdegrees = [] # empty lists
3 for i in range(n):
4    Cdegrees.append(-5 + i*0.5)
5    Fdegrees.append((9.0/5)*Cdegrees[i] + 32)
```

Python tiene una forma compacto de construir listas, usando bucles for:

```
1 Cdegrees = [-5 + i*0.5 for i in range(n)]
2 Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
```

Forma general

```
1 algunaLista = [expresion for elemento in algunaLista]
```

Demostracion interactiva de enumeracion de listas

```
1 n = 4
2 Cdegrees = [-5 + i*2 for i in range(n)]
3 Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
```

▸ Visualizacion de le ejecucion

Listas anidadas : Listas de listas

- Una lista puede contener cualquier objeto, inclusive otra lista.
- En lugar de almacenar una tabla como dos listas separadas (una para cada columna) podemos unir las dos listas en una nueva lista.

```
1 Cdegrees = range(-20, 41, 5)
2 Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
3 table1 = [Cdegrees, Fdegrees] # lista de dos listas
4 print (table1[0]) # la lista Cdegrees
5 print ()
6 print (table1[1]) # la lista Fdegrees
7 print()
8 print (table1[1][2]) # El 3er elemento en Fdegrees
```

Columnas vs. Filas

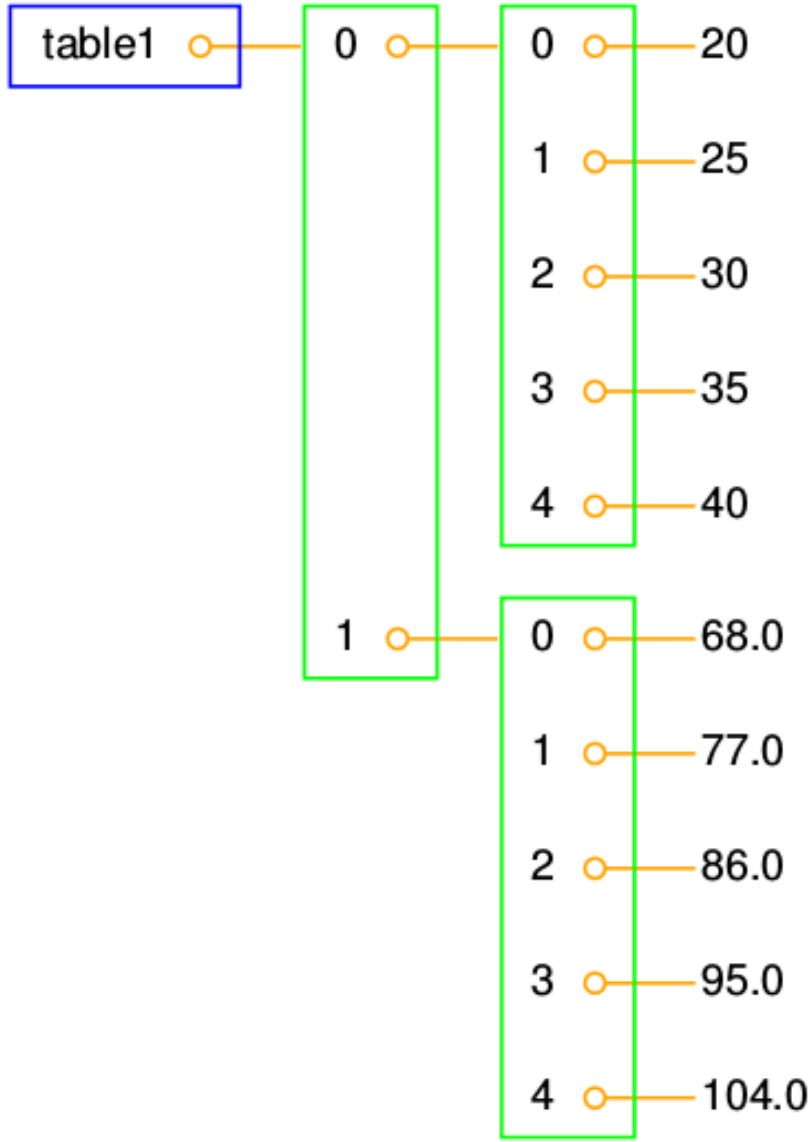
- En el slide anterior, `tabla1` es una tabla de dos columnas.
- Hagamos una tabla de filas, cada fila es un par [C,F]

```
table2 = []
for C, F in zip(Cdegrees, Fdegrees):
row = [C, F]
table2.append(row)
# mas compacyto con enumeracion de lista
table2 = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
```

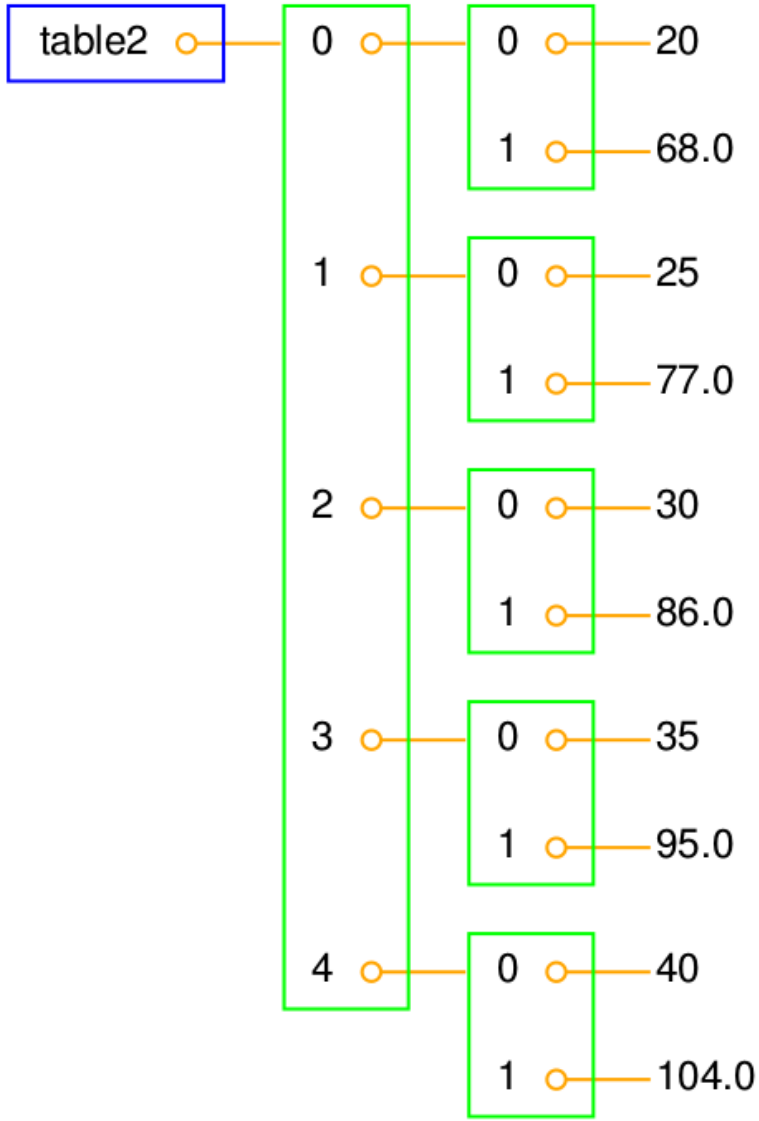
Iteracion sobre una lista anidad :

```
for C, F in table2:
    # tranaja con C y F desde una fila en tabla2
# o
for row in table2:
    C, F = row
    ...
```

Ilustracion de una tabla en columnas



Ilustracion de una tabla en filas



Extrayendo sublistas (slices)

Podemos tomar facilmente partes de una lista

```
In [83]: A = [2, 3.5, 8, 10]

In [84]: A[2:] # desde el indice 2 hasta el final de la lista
Out[84]: [8, 10]

In [85]: A[1:3] # desde el indice 1 hasta, sin incluir, el
↪  indice 3
Out[85]: [3.5, 8]

In [86]: A[:3] # desde el inicio hasta, sin incluir, el indice 3
Out[86]: [2, 3.5, 8]

In [88]: A[1:-1] # desde el indice 1 hasta ,sin incluir,el
↪  ultimo elemento
Out[88]: [3.5, 8]

In [89]: A[:] # la lista completa
Out[89]: [2, 3.5, 8, 10]

# Nota que las sublistas (slices) son copias
# de la lista original
```

Que hace el siguiente fragmento de codigo ?

```
1 Cdegrees = range(-20, 41, 5)
2 Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
3 table2 = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
4 for C, F in table2[Cdegrees.index(10):Cdegrees.index(35)]:
5    print('%5.0f %5.1f' % (C, F))
```

- Es un bucle for sobre una sublista de table2.
- Indices de la sublista :
  `Cdegrees.index(10):Cdegrees.index(35)`, es decir, los indices corresponden a los elementos 10 y (sin incluir) 35

```
1    10  50.0
2    15  59.0
3    20  68.0
4    25  77.0
5    30  86.0
```

Que hace el siguiente fragmento de codigo ?

```
1 Cdegrees = range(-20, 41, 5)
2 Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
3 table2 = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
4 for C, F in table2[Cdegrees.index(10):Cdegrees.index(35)]:
5     print('%5.0f %5.1f' % (C, F))
```

- Es un bucle for sobre una sublista de table2.
- Indices de la sublista :
  `Cdegrees.index(10):Cdegrees.index(35)`, es decir, los indices corresponden a los elementos 10 y (sin incluir) 35

```
1    10  50.0
2    15  59.0
3    20  68.0
4    25  77.0
5    30  86.0
```

Listas anidadas (I)

La idea básica de una lista anidada es que usted tiene, esencialmente, una lista compuesta de listas. Por ejemplo:

```
1 L1 = [1,2]
2 L2 = [[1,2,3],[4,5,6]]
```

Del ejemplo anterior, podemos ver que L1[0] es igual a 1, y L1[1] es igual a 2. En una lista anidada, como L2, el primer elemento de la lista, es una lista sí mismo. Entonces, L2[0] es la lista de [1,2,3], y L2[1] es la lista de [4,5,6]. Comprender este punto es vital para entender el siguiente slide.

Listas anidadas (II)

Al pedir un elemento específico en una lista, decimos: L1[x], donde x es el elemento deseado. Pero, ¿qué decimos cuando queremos un elemento específico de una lista anidada? Nosotros decimos: L1[x][y]. Esto es más fácil de mostrar que de explicar, así que aquí hay un ejemplo:

```
1 L1 = [ 1, [73,89,42,32], 62, [24, 32], 99 ]
```

En este ejemplo, L1[0] tiene el valor de 1, pero L1[1] tiene el valor de [73,89,42,32]. Desde aquí, actuaremos exactamente como si estuviéramos obteniendo un elemento de una lista no anidada, con una sola diferencia. Vamos a agregar otro conjunto de corchetes. Ejemplo:

```
1 L1[1] = [73,89,42,32]
2 L1[1][0] = 73
3 L1[1][1] = 89
```

Las tuplas

Definicion de Tuplas

Tupla es una colección de objetos de Python como una lista. La secuencia de valores almacenados en una tupla puede ser de cualquier tipo, y están indexados por números enteros. La diferencia importante entre una lista y una tupla es que las tuplas son inmutables. Los valores de una tupla están separados sintácticamente por "comas". Aunque no es necesario, es más común definir una tupla cerrando la secuencia de valores entre paréntesis. Esto ayuda a entender las tuplas de Python más fácilmente.

Las tuplas son inmutables y, por lo general, contienen una secuencia de elementos heterogéneos a los que se accede mediante desempaquetado o indexación (o incluso por atributo en el caso de tuplas con nombre).

Otro tipo de dato : Las tuplas

Las tuplas son listas constantes (no pueden **mutar o cambiar**)

```
1  t = (2, 4, 6, 'temp.pdf') # definimos una tupla
2  t = 2, 4, 6, 'temp.pdf' # podemos obviar los parentesis
3
4  t[1] = -1
5  Traceback (most recent call last):
6     File "<ipython-input-99-593c03edf054>", line 1, in <module>
7     t[1] = -1
8     TypeError: 'tuple' object does not support item assignment
9
10 t.append(0)
11 Traceback (most recent call last):
12    File "<ipython-input-100-027f59be7fb0>", line 1, in <module>
13    t.append(0)
14    AttributeError: 'tuple' object has no attribute 'append'
15
16
17 del t[1]
18 Traceback (most recent call last):
19    File "<ipython-input-101-77cea8bc7ee1>", line 1, in <module>
20    del t[1]
21    TypeError: 'tuple' object doesn't support item deletion
```

Nota sobre las tuplas (I)

La creación de tuplas en Python sin el uso de paréntesis se conoce como **Tuple Packing**.

```
# Creacion de una tupla vacia
Tuple1 = ()
print("Tupla inicial vacia: ")
print (Tuple1)

# Creacion de una tupla con el uso de cadenas de caracteres
Tuple1 = ('novato', 'data')
print("\nTupla con el uso de cadenas de caracteres (Strings) ")
print(Tuple1)

# Creando una tupla con el uso de listas
list1 = [1, 2, 4, 5, 6]
print("\nTupla usando listas []: ")
print(tuple(list1))
```

Nota sobre las tuplas (II)

```python
# Creación de una tupla con el uso de la función incorporada
Tuple1 = tuple('Simulacion')
print("\nTupla con el uso de una funcion")
print(Tuple1)

# Creacion de una tupla con datos heterogeneos
Tuple1 = (1, 'CTIC', 8, 'UNI')
print("\nTupla con datos heterogeneos ")
print(Tuple1)

# Creacion de una tupla anidada
Tuple1 = (1, 0, 1, "A")
Tuple2 = ('Data', 'Science')
Tuple3 = (Tuple1, Tuple2)
print("\nTupla anidada: ")
print(Tuple3)

# Creacion de una  tupla con repeticiones
Tuple1 = ('Geeks',) * 3
print("\nTuple with repetition: ")
print(Tuple1)
```

Tuplas : listas inmutables

Las tuplas pueden hacer mucho de lo que las listas pueden hacer:

```
t = t +(-1.0, -2.0) # juntamos dos tuplas
t
t[1] # son indexadas
t[2:] #subtuplas (slices)
6 in t # pertenencia
```

Funcionalidad : Tuplas vs. Listas

- Las tuplas son inmutables (constantes) y por lo tanto estan protegidas de cambios accidentales.
- Las tuplas son mas rapidas que las listas.
- Las tuplas son ampliamente utilizadas en software desarrollado con python.
- Las tuplas (pero no las listas) se pueden usar como claves (keys) en otra estructura de dato llamada diccionarios.

Concatenación de tuplas

La concatenación de la tupla es el proceso de unión de dos o más tuplas. La concatenación se realiza mediante el uso del operador "+". La concatenación de las tuplas se realiza siempre desde el final de la tupla original. Otras operaciones aritméticas no se aplican en las tuplas.

Nota: solo se pueden combinar los mismos tipos de datos con la concatenación; se produce un error si se combinan una lista y una tupla.

```
1 # Concatenacion de tuplas
2 Tuple1 = (0, 1, 2, 3)
3 Tuple2 = ('Math', 'For', 'DS')
4 Tuple3 = Tuple1 + Tuple2
5
6 # IMprimiendo la Tupla3
7 print("\nTuplas despues de la concatenacion: ")
8 print(Tuple3)
```

Slicing de tuplas

El slicing de una tupla se realiza para obtener un rango específico o una porción de subelementos de una tupla. El slicing también se puede hacer a listas y matrices. La indexación en una lista da como resultado la obtención de un único elemento, mientras que la opción de segmentación permite obtener un conjunto de elementos.

```
Tuple1 = tuple('DATASCIENCEFORBUSINESS')

# Removiendo el primer elemento
print("Removal of First Element: ")
print(Tuple1[1:])

# Invirtiendo el orden
print("\nOrden invertido de los elementos de la tupla: ")
print(Tuple1[::-1])

# imprimiendo los elementos en un rango determinado
print("\nElementos entre 4-9: ")
print(Tuple1[4:9])
```

Ejercicio 6

En una baraja de cartas, cada carta es una combinación de un rango y un palo. Hay 13rangos: as (A), 2, 3, 4, 5, 6, 7, 8, 9, 10, jack (J), reina (Q), rey (K) y cuatro símbolos: palos (C), diamantes (D), corazones (H) y espadas (S). Una tarjeta típica puede ser D3. Escriba las sentencias que generen un mazo de cartas, es decir, todas las combinaciones CA, C2, C3, etc. a SK.

Ejercicio 7

Calcular combinaciones. Considere un número de identificación que consta de dos letras y tres dígitos, por ejemplo, RE198. Cómo podemos tener todas las combinaciones diferentes, y cómo puede un programa generar todos estos combinaciones?

Si una colección de n cosas puede tener m1 variaciones de la primera cosa, m2 de la segunda y así sucesivamente, el número total de variaciones de la colección es igual a m1 * m 2 ... * mn.En particular, el número de ID ejemplificado anteriormente puede tener 26 * 26 * 10 * 10 * 10 = 676000 variaciones Para generar todas las combinaciones, debemos tener cinco bucles anidados.Los dos primeros pasan sobre todas las letras A, B, y así sucesivamente hasta Z, mientras que los tres siguientes pasan sobretodos los dígitos 0; 1; ... ; 9.

Nota : Use `string.ascii_uppercase` perteneciente al modulo `string`

Estructura de Decisión : IF

El flujo de un programa a veces necesita tomar decisiones en base a evaluaciones booleanas, esto se traduce en una estructura condicional. Para ver esto de manera matemática, veamos la siguiente función :

$$f(x) = \begin{cases} \sin(x) & , \quad 0 \leq x \leq \pi \\ 0 & , \quad \text{en otro caso} \end{cases}$$

Estructura de Decisión : IF

En python la implementación de esta función necesitaría la evaluación del valor de x, esto lo resolvemos con la estructura `if`.

```
1 from math import pi, sin
2 def f(x):
3     if 0 <= x <= pi:
4         value = sin(x)
5     else:
6         value = 0
7     return value
8
9 f(2)
```

Estructura de Decisión : Bloques IF-ELSE

La estructura general de un bloque if-else es:

```
1 if condition:
2     <bloque de sentencias ejecutadas si condicion es TRUE>
3 else:
4     <bloque de sentencias ejecutadas si condicion es FALSE>
```

Estructura de Decisión : Bloques IF-ELSE

Con la palabra reservada `elif`, abreviatura de `else if`, podemos tener varias condicionales `if` mutuamente excluyentes, que permiten una bifurcación múltiple del flujo del programa:

```
1  if condition1:
2     <bloque de sentencias>
3  elif condition2:
4     <bloque de sentencias>
5  elif condition3:
6     <bloque de sentencias>
7  else:
8     <bloque de sentencias>
9
10 <siguiente sentencia , fuera de los IF>
```

Otro tipo de dato : Los diccionarios

El presente tema aborda muchas técnicas para interpretar información en archivos y almacenar los datos en objetos prácticos de Python para un análisis de datos. Un objeto particularmente útil para muchos propósitos es el diccionario, que mapea objetos en objetos, muy a menudo se conecta a varios tipos de datos que luego se pueden buscar a través de las cadenas (strings). Hasta ahora hemos almacenado información en varios tipos de objetos, como números, cadenas, listas, tuplas y matrices. Un diccionario es un objeto muy flexible para almacenar diverso tipo de información, y en particular al leer archivos. Por lo tanto, es hora para introducir el diccionario.

Temperaturas en ciudades

```
1 # cada elemento se escribe de la forma llave:valor
2 temps = {'Oslo': 13, 'London': 15.4, 'Paris': 17.5}
3 # o
4 temps = dict(Oslo=13, London=15.4, Paris=17.5)
```

Agregar nuevos elementos

```
1 temps['Madrid'] = 26.0
2
3 print(temps)
```

Diccionarios : Operaciones

La cadena `city` hace las veces del indice en una estructura vectorial (como las listas, tuplas, vectores , etc) y para acceder al valor lo hacemos de manera simular a las anteriores estructuras vectoriales.

For para barrer todos los valores por medio de las llaves

```
1 for city in temps:
2     print('La temperatura en %s is %g' % (city, temps[city]))
```

Podemos verificar si una llave (key) esta presente en una variable de tipo diccionario haciendo uso de una estructura de decision `IF`

If para verificar

```
1 if 'Berlin' in temps:
2     print('Berlin:', temps['Berlin'])
3 else:
4     print('NO hay datos para Berlin')
```

Diccionarios : Operaciones

Operaciones booleanas con diccionarios

```
1 'Lima' in temps
2 # debe retornar False como resultado de la consulta
```

Las llaves (keys) y los valores pueden ser extraidos como listas a partir del diccionario.

```
1 temps.keys()
2
3 temps.values()
```

Diccionarios : Operaciones

Se puede ordenar las llaves

```
# podemos mostrar como se creo el diccionario :
for city in temps:
    print(city)

# o mostrar las llaves de manera ordenada :
for city in sorted(temps):
    print(city)
```

Borrar un par clave:valor (Comando **del**)

```
len(temps)
del temps['Oslo']
print(temps)
len(temps)
```

Diccionarios : Operaciones

Copia de diccionarios

```
temps_copy = temps.copy()

del temps_copy['Paris']
# esto no afecta el diccionarios original :temps

temps_copy

temps
```

Si dos variables hacen referencia al mismo diccionario (Copia sin usar el metodo .copy)

```
t1 = temps
t1['Stockholm'] = 10.0 # cambiamos t1
t1
temps # temps tambien cambio
```

Primalidad

```
limit = 121


flags = {}
for i in range(2,limit+1):
    flags[i] = True

p = 2
while p <= limit:
    if flags[p] == True:
        print(p)
        m = p*p
        while m <= limit:
            flags[m] = False
            m += p
    p += 1

print(flags)
```

Diccionarios : Operaciones

Primalidad

```
1  from numpy import arange
2  max=12000
3  checked={}
4  primes =[]
5  numbers=[]
6
7  for i in arange(2,max):
8     numbers.append(i)
9     checked[i] = False
10
11 for i in numbers:
12    if checked[i] == True:
13       continue
14    else:
15       primes.append(i)
16       checked[i] = True
17       for x in numbers:
18          if x%i==0 and checked[x]==False:
19             checked[x]=True
20             print(x)
21
22 print(primes)
```